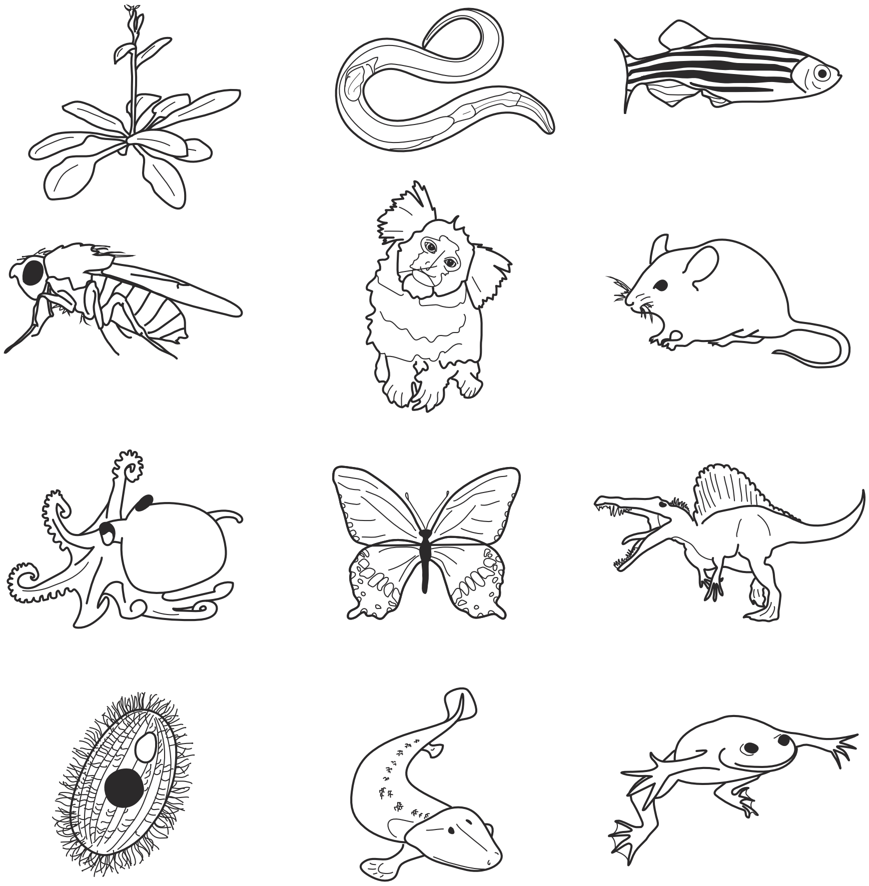
THE UNIVERSITY OF
CHICAGO BIOLOGICAL SCIENCES

BSD qBio^8 Boot Camp



September 8–14, 2022

Group logos by Professor Stephanie Palmer

BSD qBio^8

General Schedule

Thursday, September 8

10:30-11:00	Check-in with TAs at BSLC lobby
11:00-11:20	Welcome to qBio8 - Novembre Carbonetto Hall
11:20-12:00	Team-building activities
12:00-1:30	Lunch and group compacts
1:30-3:00	Basic/Advanced comp. I
3:00-3:30	Coffee break
3:30-5:00	Basic/Advanced comp. I

Friday, September 9

8:30-10:00	Basic/Advanced comp. II
10:00-10:30	Coffee break
10:30-12:00	Basic/Advanced comp. II
12:00-2:30	Lunch and DuSable African American Museum Tour
2:30-2:55	Welcome to UChicago Biosciences - Dean Kovar
3:00-4:30	Professional development with TAs
4:30-4:45	Coffee break
4:45-5:30	Graduate Student Research Talks

Monday, September 12

8:30-10:00	Data visualization - Carbonetto (*C.jacchus D.rerio P.polytes T.thermophila*) Defensive programming - Hernandez (*A.thaliana C.elegans T.roseae X.laevis*) Statistics for a data rich world - L. Chen (*D.melanogaster M.musculus O.bimaculoides S.aegyptiacus*)
10:00-10:30	Coffee break
10:30-12:00	Data visualization - Carbonetto (*C.jacchus D.rerio P.polytes T.thermophila*) Defensive programming - Hernandez (*A.thaliana C.elegans T.roseae X.laevis*) Statistics for a data rich world - L. Chen (*D.melanogaster M.musculus O.bimaculoides S.aegyptiacus*)
12:00-1:30	Lunch
1:30-3:00	Data visualization - Carbonetto (*A.thaliana M.musculus O.bimaculoides X.laevis*) Defensive programming - Hernandez (*D.melanogaster D.rerio S.aegyptiacus T.thermophila*) Statistics for a data rich world - L. Chen (*C.elegans C.jacchus P.polytes T.roseae*)
3:00-3:30	Coffee break
3:30-5:00	Data visualization - Carbonetto (*A.thaliana M.musculus O.bimaculoides X.laevis*) Defensive programming - Hernandez (*D.melanogaster D.rerio S.aegyptiacus T.thermophila*) Statistics for a data rich world - L. Chen (*C.elegans C.jacchus P.polytes T.roseae*)

Tuesday, September 13

8:30-10:00	Data visualization - Carbonetto (*C.elegans D.melanogaster S.aegyptiacus T.roseae*) Defensive programming - Hernandez (*C.jacchus M.musculus O.bimaculoides P.polytes*) Statistics for a data rich world - L. Chen (*A.thaliana D.rerio T.thermophila X.laevis*)
10:00-10:30	Coffee break
10:30-12:00	Data visualization - Carbonetto (*C.elegans D.melanogaster S.aegyptiacus T.roseae*) Defensive programming - Hernandez (*C.jacchus M.musculus O.bimaculoides P.polytes*)

	Statistics for a data rich world - L. Chen (*A.thaliana D.rerio T.thermophila X.laevis*)
12:00-1:30	Lunch
1:30-3:00	Workshop - Immunology - Khan (*A.thaliana C.elegans D.rerio X.laevis*)
	Workshop - Population genetics - Berg (*D.melanogaster M.musculus S.aegyptiacus T.roseae*)
3:00-3:30	Coffee break
3:30-5:00	Workshop - Immunology - Khan (*A.thaliana C.elegans D.rerio X.laevis*)
	Workshop - Population genetics - Berg (*D.melanogaster M.musculus S.aegyptiacus T.roseae*)
	Workshop - Transcriptomics - M. Chen (*C.jacchus O.bimaculoides P.polytes T.thermophila*)

Wednesday, September 14

8:30-10:00	Workshop - Immunology - Khan (*C.jacchus D.melanogaster T.roseae T.thermophila*)
	Workshop - Population genetics - Berg (*A.thaliana C.elegans O.bimaculoides P.polytes*)
	Workshop - Transcriptomics - M. Chen (*D.rerio M.musculus S.aegyptiacus X.laevis*)
10:00-10:30	Coffee break
10:30-12:00	Workshop - Immunology - Khan (*C.jacchus D.melanogaster T.roseae T.thermophila*)
	Workshop - Population genetics - Berg (*A.thaliana C.elegans O.bimaculoides P.polytes*)
	Workshop - Transcriptomics - M. Chen (*D.rerio M.musculus S.aegyptiacus X.laevis*)
12:00-1:30	Lunch
1:30-3:00	Workshop - Immunology - Khan (*M.musculus O.bimaculoides P.polytes S.aegyptiacus*)
	Workshop - Population genetics - Berg (*C.jacchus D.rerio T.thermophila X.laevis*)
	Workshop - Transcriptomics - M. Chen (*A.thaliana C.elegans D.melanogaster T.roseae*)
3:00-3:30	Coffee break
3:30-5:00	Workshop - Immunology - Khan (*M.musculus O.bimaculoides P.polytes S.aegyptiacus*)
	Workshop - Population genetics - Berg (*C.jacchus D.rerio T.thermophila X.laevis*)
	Workshop - Transcriptomics - M. Chen (*A.thaliana C.elegans D.melanogaster T.roseae*)
5:15-5:45	Wrap-up
5:45-7:30	Reception with Dean's Council Students

Materials

Opening Tutorials:

Emphasize basic and advanced computing concepts in R

1. Basic computing I
2. Basic computing II
3. Advanced computing

Tutorials:

Emphasize general challenges in biological data science

4. Data visualization
5. Defensive Programming
6. Statistics for a data-rich world

Workshops:

Expand upon and apply techniques and ideas from the tutorials in the context of specific disciplinary challenge areas

7. RNAseq Workshop
8. Population Genetics Workshop
9. Immunology Workshop

Base R
Cheat Sheet

Getting Help

Accessing the help files

?mean
Get help of a particular function.

help.search(‘weighted mean’)
Search the help files for a word or phrase.

help(package = ‘dplyr’)
Find help for a package.

More about an object

str(iris)
Get a summary of an object’s structure.

class(iris)
Find the class an object belongs to.

Using Packages

install.packages(‘dplyr’)
Download and install a package from CRAN.

library(dplyr)
Load the package into the session, making all its functions available to use.

dplyr::select
Use a particular function from a package.

data(iris)
Load a built-in dataset into the environment.

Working Directory

getwd()
Find the current working directory (where inputs are found and outputs are sent).

setwd(‘C://file/path’)
Change the current working directory.

Use projects in RStudio to set the working directory to the folder you are working in.

Vectors

Creating Vectors

`c(2, 4, 6)`	`2 4 6`	Join elements into a vector
`2:6`	`2 3 4 5 6`	An integer sequence
`seq(2, 3, by=0.5)`	`2.0 2.5 3.0`	A complex sequence
`rep(1:2, times=3)`	`1 2 1 2 1 2`	Repeat a vector
`rep(1:2, each=3)`	`1 1 1 2 2 2`	Repeat elements of a vector

Vector Functions

sort(x)
Return x sorted.

table(x)
See counts of values.

rev(x)
Return x reversed.

unique(x)
See unique values.

Selecting Vector Elements

By Position

`x[4]`	The fourth element.
`x[-4]`	All but the fourth.
`x[2:4]`	Elements two to four.
`x[-(2:4)]`	All elements except two to four.
`x[c(1, 5)]`	Elements one and five.

By Value

`x[x == 10]`	Elements which are equal to 10.
`x[x < 0]`	All elements less than zero.
`x[x %in% c(1, 2, 5)]`	Elements in the set 1, 2, 5.

Named Vectors

`x[‘apple’]`	Element with name ‘apple’.

Programming

For Loop

```
for (variable in sequence){

   Do something

}
```

Example

```
for (i in 1:4){

   j <- i + 10

   print(j)

}
```

While Loop

```
while (condition){

   Do something

}
```

Example

```
while (i < 5){

   print(i)

   i <- i + 1

}
```

If Statements

```
if (condition){
   Do something
} else {
   Do something different
}
```

Example

```
if (i > 3){
   print(‘Yes’)
} else {
   print(‘No’)
}
```

Functions

```
function_name <- function(var){

   Do something

   return(new_variable)
}
```

Example

```
square <- function(x){

   squared <- x*x

   return(squared)
}
```

Reading and Writing Data

Also see the **readr** package.

Input	Ouput	Description
`df <- read.table(‘file.txt’)`	`write.table(df, ‘file.txt’)`	Read and write a delimited text file.
`df <- read.csv(‘file.csv’)`	`write.csv(df, ‘file.csv’)`	Read and write a comma separated value file. This is a special case of read.table/ write.table.
`load(‘file.RData’)`	`save(df, file = ’file.Rdata’)`	Read and write an R data file, a file type special for R.

Conditions

`a == b`	Are equal	`a > b`	Greater than	`a >= b`	Greater than or equal to	`is.na(a)`	Is missing
`a != b`	Not equal	`a < b`	Less than	`a <= b`	Less than or equal to	`is.null(a)`	Is null

RStudio® is a trademark of RStudio, Inc. • CC BY Mhairi McNeill • mhairihmcneill@gmail.com

Learn more at web page or vignette • package version • Updated: 3/15

Types

Converting between common data types in R. Can always go from a higher value in the table to a lower value.

`as.logical`	`TRUE, FALSE, TRUE`	Boolean values (TRUE or FALSE).
`as.numeric`	`1, 0, 1`	Integers or floating point numbers.
`as.character`	`'1', '0', '1'`	Character strings. Generally preferred to factors.

`as.factor`	`'1', '0', '1',`
	`levels: '1', '0'`
		Character strings with preset levels. Needed for some statistical models.

Maths Functions

`log(x)`	Natural log.	`sum(x)`	Sum.
`exp(x)`	Exponential.	`mean(x)`	Mean.
`max(x)`	Largest element.	`median(x)`	Median.
`min(x)`	Smallest element.	`quantile(x)`	Percentage quantiles.
`round(x, n)`	Round to n decimal places.	`rank(x)`	Rank of elements.
`signif(x, n)`	Round to n significant figures.	`var(x)`	The variance.
`cor(x, y)`	Correlation.	`sd(x)`	The standard deviation.

Variable Assignment

```
> a <- 'apple'
> a
[1] 'apple'
```

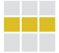
The Environment

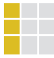
`ls()`	List all variables in the environment.
`rm(x)`	Remove x from the environment.
`rm(list = ls())`	Remove all variables from the environment.

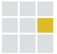
You can use the environment panel in RStudio to browse variables in your environment.

Matrices

`m <- matrix(x, nrow = 3, ncol = 3)`
Create a matrix from x.


`m[2, ]` - Select a row


`m[ , 1]` - Select a column


`m[2, 3]` - Select an element

`t(m)`
Transpose

`m %*% n`
Matrix Multiplication

`solve(m, n)`
Find x in: m * x = n

Lists

`l <- list(x = 1:5, y = c('a', 'b'))`
A list is a collection of elements which can be of different types.

`l[[2]]`	`l[1]`	`l$x`	`l['y']`
Second element of l.	New list with only the first element.	Element named x.	New list with only element named y.

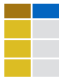
Also see the **dplyr** package.

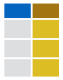
Data Frames

`df <- data.frame(x = 1:3, y = c('a', 'b', 'c'))`
A special case of a list where all elements are the same length.

x	y
1	a
2	b
3	c

List subsetting

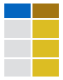
`df$x`


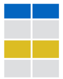
`df[[2]]`


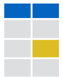
Understanding a data frame

`View(df)`	See the full data frame.
`head(df)`	See the first 6 rows.

Matrix subsetting

`df[ , 2]`


`df[2, ]`


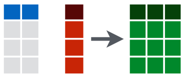
`df[2, 2]`


`nrow(df)`
Number of rows.

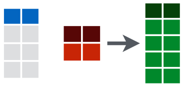
`ncol(df)`
Number of columns.

`dim(df)`
Number of columns and rows.

cbind - Bind columns.



rbind - Bind rows.



Strings

Also see the **stringr** package.

`paste(x, y, sep = ' ')`	Join multiple vectors together.
`paste(x, collapse = ' ')`	Join elements of a vector together.
`grep(pattern, x)`	Find regular expression matches in x.
`gsub(pattern, replace, x)`	Replace matches in x with a string.
`toupper(x)`	Convert to uppercase.
`tolower(x)`	Convert to lowercase.
`nchar(x)`	Number of characters in a string.

Factors

`factor(x)`
Turn a vector into a factor. Can set the levels of the factor and the order.

`cut(x, breaks = 4)`
Turn a numeric vector into a factor by 'cutting' into sections.

Statistics

`lm(y ~ x, data=df)`
Linear model.

`glm(y ~ x, data=df)`
Generalised linear model.

`summary`
Get more detailed information out a model.

`t.test(x, y)`
Perform a t-test for difference between means.

`pairwise.t.test`
Perform a t-test for paired data.

`prop.test`
Test for a difference between proportions.

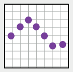
`aov`
Analysis of variance.

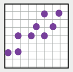
Distributions

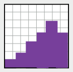
	Random Variates	Density Function	Cumulative Distribution	Quantile
Normal	`rnorm`	`dnorm`	`pnorm`	`qnorm`
Poisson	`rpois`	`dpois`	`ppois`	`qpois`
Binomial	`rbinom`	`dbinom`	`pbinom`	`qbinom`
Uniform	`runif`	`dunif`	`punif`	`qunif`

Plotting

Also see the **ggplot2** package.


`plot(x)`
Values of x in order.


`plot(x, y)`
Values of x against y.


`hist(x)`
Histogram of x.

Dates

See the **lubridate** package.

RStudio® is a trademark of RStudio, Inc. • CC BY Mhairi McNeill • mhairihmcneill@gmail.com • 844-448-1212 • rstudio.com

Learn more at web page or vignette • package version • Updated: 3/15

Basic Computing 1 — Introduction to R, Part 1[*]

Peter Carbonetto and Aarti Venkat *University of Chicago*

The aim of this workshop is to introduce R and RStudio, and to use R to analyze data interactively. We will focus on one particularly widely used data structure, the *data frame.* We will learn how to import tabular data into a data frame, and we will learn how to inspect, manipulate and analyze the data frame. This workshop is intended for biologists with little to no background in programming.

About this tutorial

In this tutorial, we will learn about the basic elements of R through analysis of a small data set. You are welcome to try this tutorial on your own, before or after the in-class tutorial. To try the examples before the data have been collected in class, you may use the data from previous years, stored in files `laptops_2019.csv`, `laptops_2020.csv` and `laptops_2021.csv`. To use these data, make a copy of one of these files, and call it `laptops.csv`.

Do I have what I need?

To follow the examples in this tutorial, you will need to install R and RStudio. To install R, follow instructions at `cran.rstudio.com`. Then install Rstudio following the instructions at `bit.ly/2JARd4v`.

From GitHub, download the tutorial materials to your computer, and make sure you know where to find them.

Now you can launch RStudio.

Where am I?

Make sure your R working directory in is the same directory containing the tutorial materials; you can run `getwd()` and `list.files()` to check this. (If you need to change your R working directory, you can either use the `setwd()` function or, in RStudio, you can select **Session > Set Working Directory > Choose Directory. . . .**)

Quit applications that are not needed and other "clutter" to reduce distractions. *Remember, the computer is device of distraction.*

Also, it is best if you start with a fresh workspace; you can refresh your environment by selecting **Session > Clear Workspace** from the RStudio menu.

[*]This document is included as part of the Basic Computing 1—Introduction to R tutorial packet for the BSD qBio Bootcamp, University of Chicago, 2022. **Current version**: August 11, 2022; **Corresponding author**: pcarbo@uchicago.edu. Thanks to Stefano Allesina, John Novembre, Stephanie Palmer and Matthew Stephens for their guidance.

A data-centric view of programming

There are many programming languages. Some of you may have experience with one of them, or more than one. R belongs to the modern family of *interactive programming languages* (e.g., Python, MATLAB, Julia), and was built on the revolutionary programming language Scheme developed at MIT. R inspires a "data-centric" view of programming, in which we focus on *data structures* (the objects) rather than procedures.

A brief tour of RStudio

For this introductory tutorial, we will use RStudio. It is an Integrated Development Environment (IDE) for R. One advantage of RStudio is that the environment will look identical irrespective of your computer architecture (Windows, Mac, Linux). RStudio also has many features that make writing code easier.

The main RStudio interface is split up into "panels", including:

1. **Console:** This is a panel containing an instance of R. For this tutorial, we will work mainly in this panel.
2. **Source:** In this panel, you can write your code and save it to a file. The code and also be run from this panel, but the actual results show up in the console.
3. **Environment:** This panel lists all the variables (objects) you created in your R environment.
4. **History:** This gives you the history of the commands you typed.
5. **Plots:** This panel shows you all the plots you drew.

Other tabs allow you to access the list of packages you have loaded, and the help page for commands (e.g., type `help(p.adjust)` in the console).

Activity: Collaborative data collection

To perform a data analysis, we need a data set to analyze. We will collect the data collaboratively.

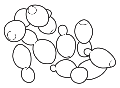
Specifically, we will collect data about the laptops of incoming BSD graduate students. Once we have collected the data, we will write some R code to answer basic questions such as: what is the most common operating system among the population (where "population" is the cohort of incoming BSD grad students)?

We will collect the data in the Etherpad. We will record four data points: first/given name, operating system, amount of memory (in GB), and number of processors (CPUs). We will enter these data into a table with four columns. We will use commas to separate the columns in the table. The first few lines of the table will look something like this:

```
name,os,mem,cpus
rishi,mac,8,4
anna,windows,4,1
peter,mac,16,4
```

Ask your teammates or neighbors for help if you are unsure how to find this information about your laptop.

Once we have collected the data from the entire class, I will create a CSV file, `laptops.csv`, and share it with you on Box.



S. cerevisiae

Import the data into R

You should now have a CSV file on your computer, `laptops.csv`, containing the data we collected. (What is CSV short for?) The basic R function for importing data from a CSV file is called `read.csv`:

```
laptops <- read.csv("laptops.csv",comment = "#",stringsAsFactors = FALSE)
```

In time, we will understand what this code is doing. For now, we will jump ahead.

This command will only work if your R working directory is the same as the directory containing `laptops.csv`. You can check your working directory with `getwd()` and `list.files()`. If you are in the wrong directory, use **Session > Set Working Directory** from the RStudio menu bar to move to the right directory.

Whenever you create an object, you need to give it a name. A good name reminds you of its contents and its purpose.

Assuming you started with a clean workspace, your workspace, or "environment", should now contain a single object, `laptops`:

```
ls()
```

This object stores the output of our `read.csv` call. Invoking its name will print its contents:

```
laptops
```

Or you can explicitly type

```
print(laptops)
```

which does the same thing.

What *kind* of object is it?

```
class(laptops)
```

It is a *data frame*. This is R's way of saying "tabular data" or "spreadsheet".

If you are used to the "point-and-click" way of doing things in software such as Excel, it may seem burdensome to have arrived at this point where we have written a bunch of code, and all we have done is printed the contents of the table to the screen. But now that we have created a data frame, there are many things we can do it.

Note the advantages of R are less striking when working with a small data set. We will work with a larger data set in Part 2.

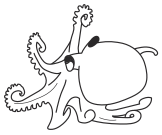
Inspect the data

R has many commands that are easy to use and quickly give you insight into the data. Let's try some of the more commonly used commands for inspecting and summarizing the contents of a data frame.

Write your code here.

Some of these commands, like `head` and `summary`, are "generic" functions, meaning that they work for many types of objects.

Now we will proceed to learn about R by trying to answer some basic questions about laptops. The first questions can be answered with very simple code, and then the code will get progressively more complicated.



O. bimaculoides

Simple questions

What is the largest and smallest amount of memory?

Write your code here.

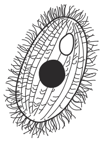
What is the most and least number of CPUs?

Write your code here.

What is the most common CPU amount, the most common memory amount, and the most common combi-

nation of CPUs and memory?

Write your code here.



T. thermophila

More difficult questions

Does anyone in Basic Computing 1 share the same name?

Write your code here.

Are all the laptops multicore (more than one processor)?

Write your code here.

Who has the most CPUs?

Write your code here.

How many different OSs are there, and which OS is used most?

Write your code here.

Observe that both assignment (creating new objects) and overwriting both use <-. So be careful—*there is no undo command in R!*

This ability to overwrite and have multiple data objects floating around in your environment underscores the importance of *naming your objects well* and *keeping track of what is in your environment*; names such as “x”, “y”, “temp” or “data”, while commonly used, should only be used for temporary or “throwaway” calculations. It isn’t unusual to have half a dozen different copies of a data set over the course of an analysis. Another important practice is to document your code and periodically save your results in case you do accidentally overwrite something important!

Among Mac laptops (or Windows laptops), what is the average amount of memory?

Write your code here.

Hypothesis: Windows laptops tend to have more memory than Mac laptops. *True or false?*

Write your code here.

Save your work

This is a good time to save our work. To save your results, go to **Session > Save Workspace As** in RStudio, or run

```
save.image("basic_computing_1.RData")
```

Later, to restore your environment, select **Session > Load Workspace** in RStudio, or run

```
load("basic_computing_1.RData")
```

What is the difference between **File > Save As** *and* **Session > Save Workspace As**?

C. jacchus

Deconstructing the data frame

Let's now take a second look at the laptops data frame, with a greater intention toward understanding R.

The data frame is an example of a *composite data structure*—that is, it is made of simpler data objects. These "atomic" data objects are R's "building blocks" for all other data structures. R has a very large variety of composite data structures, but the data frame may be the most popular.

In a data frame, the *columns are the atomic data objects*; in the next sections, we will closely examine the individual columns of the laptops data frame.

Text data

Let's begin with the "os" column:

```
x <- laptops$os
print(x)
```

In R, the "character" type is used to store text data:

```
class(x)
```

You can access individual elements by their index: the first element is indexed at 1, the second at 2, *etc.*

```
x[1]
x[2]
```

R has many built-in functions for operating on text data. Here are some examples:

```
nchar(x)
toupper(x)
sort(x)
unique(x)
paste(x[1],x[2],x[3])
```

Which of these functions are specific to text data?

Just as data types can be combined to form more complex data structures, operations can also be combined (provided the combination makes sense, of course!). For example,

```
toupper(unique(x))
```

is effectively equivalent to

```
y <- unique(x)
toupper(y)
```

Numeric data

The "mem" column is an example of numeric data:

```
x <- laptops$mem
class(x)
```

Specifically, it is an integer type since all the numbers are whole numbers. There is another data type, "numeric", used to store real numbers (or rather their approximations, as computers have limited memory and thus cannot store numbers like π, or even 0.2), e.g.,

```
y <- x / 100
class(y)
```

Like the character data type, a numeric object is a vector, and individual elements can be accessed their index,

```
x[1]
x[2]
```

Also, like the character data type, R has many built-in functions for working with numbers, such as

```
sum(x)
max(x)
range(x)
sqrt(x)
cos(x)
log(x)
```

Notice that, for example, `log(x)` computes the logarithm for *each data point in* `x`, and the result is *a vector of the same length as* `x`. This allows large calculations to be accomplished very simply; for example, if `x` contains 10 million numbers, `y <- sqrt(x)` will compute the square root of these 10 million numbers, and store the result in vector `y`.

Given that R was born for statistics, there are many statistical operations—from basic to sophisticated—that you can perform on numeric data:

```
median(x)
var(x)
quantile(x)
```

Finally, standard mathematical operations such as + (addition), - (subtraction), * (multiplication), / (division), and ^ (exponentiation) can be applied to numbers.

For more complex calculations, the mathematical and statistical operations can be combined, for example,

```
y <- log(x + 0.1)
```

Logical data

The laptops data frame does not contain logical data. But we can generate logical data from one of the columns:

```
x <- laptops$os == "mac"
print(x)
class(x)
```

The logical data type takes only two values, `TRUE` and `FALSE`.

Exercise 1: Besides equality, other comparison operators include > (greater than), < (less than), `!=` (differs) and >= and <=. Write an expression using one or more of these operators to create some new logical data.

Write your code here.

Exercise 2: You can also formulate more complicated logical statements (perhaps using multiple variables or columns of a data frame) using `&` (and), `|` (or), and `!` (not). Write an expression using these operators, as well as the ones above, to create some new logical data.

Write your code here.

E. coli

Activity: what is a "factor"?

So far, we have gotten familiar with three basic data types: character, numeric and logical. There is a fourth important atomic data type in R: *the factor.* Unknowningly, we have already used a factor in the examples above.

What is unusual about factors is that there is no equivalent of factors in other popular programming languages, at least not as a native data type. And yet you will find that they are extremely useful.

None of the columns in the laptops data frame are a factor. But, like the logical data type, we can create a factor from other data types. Let's try this with both the "mem" and "os" columns:

```
x <- factor(laptops$mem)
y <- factor(laptops$os)
class(x)
class(y)
```

Now run the following lines of code:

```
print(x)
print(y)
summary(x)
summary(y)
as.numeric(x)
as.numeric(y)
```

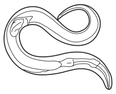
Question: Based on the results from running thiis code, what do you think a factor is?

More questions: Which representation to do you prefer for "mem", numeric or factor? What about the "os" column? Are other columns good candidates for being factors?

These questions—should I convert my data to a factor—touch on the broader question of *data representation* or *data encoding*. Choosing the right representation of your data can be important to an effective data analysis. (You might find that factors are useful in the programming challenge.)

If you prefer the “os” column to be a factor, for example, you can change it inside the data frame by doing

```
summary(laptops)
laptops$os <- factor(laptops$os)
summary(laptops)
```



C. elegans

Activity: Analyze laptop trends over time

Having gotten a bit of a feel for how to analyze data in R, we will now use the laptops data collected by BSD grad students over several years to *explore trends in laptops over time.* This is a freeform activity that will require some creativity and teamwork.

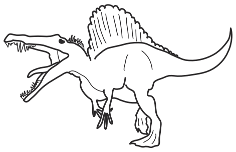
Before writing any R code, formulate some questions which will guide your analysis, such as: Did memory or number of CPUs tend to increase over time? Do more students prefer Macs now than they did in 2019? Write down a few guiding questions in the space below.

✎ *Formulate your guiding questions here.*

These next few lines of code import the laptops data from the years 2019–2021, and combine them into a single data frame. *You will need to modify this code if you want to also include the data from 2022.*

```
laptops_2019 <- read.csv("laptops_2019.csv",comment = "#",stringsAsFactors = FALSE)
laptops_2020 <- read.csv("laptops_2020.csv",comment = "#",stringsAsFactors = FALSE)
laptops_2021 <- read.csv("laptops_2021.csv",comment = "#",stringsAsFactors = FALSE)
laptops_2019$year <- 2019
laptops_2020$year <- 2020
laptops_2021$year <- 2021
laptops <- rbind(laptops_2019,laptops_2020,laptops_2021)
```

At the end of this activity we will share and discuss these analyses using the Etherpad.



S. aegyptiacus

Programming Challenge

Instructions

Work with your team to solve the following exercise. When you have found the solution, go to the `jnovembre.github.io/BSD-QBio8` and follow the link "Submit solution to challenge 1" to submit your answer.

Collaboration strategy

Before diving into the problems, first agree on a collaboration strategy with your teammates. Important aspects include communication and co-ordination practices, and setting goals and deadlines. How will your team collaborate on code, and share solutions? (Consider online resources such as Etherpad or Google Drive.) The aim is not just to complete the challenges, but also to do collaboratively; all team members should be included, and should have the opportunity to contribute and learn from each other.

Nobel nominations

The file `nobel_nominations.csv` contains data on Nobel Prize nominations from 1901 to 1964. There are three columns (the file has no "header"): (1) the field (e.g., "Phy" for physics), (2) the year of the nomination, and (3) the id and name of the nominee.

1. Take Chemistry (`Che`). Who received the most nominations?
2. Find all researchers who received nominations in more than one field.
3. Take Physics (`Phy`). Which year had the largest number of nominees?
4. For each field, what is the average number of nominees per year? Calculate the number of nominees in each field and year, and take the average across all years.

Hints

- You will need to subset the data. To make your calculations more clear, it may be helpful to give names to the columns. For example, suppose you imported the data into a data frame called `nobel`. Then `colnames(nobel) <- c("field","year","nominee")` should do the trick.
- The simplest way to obtain a count from a vector is to use the `table` function. For example, the command `sort(table(x))` produces a table of the occurrences in `x`, in which the counts are sorted from smallest to largest.
- You can also use the same function, `table`, to build a table using more than one vector. For example, suppose `x` and `y` are vectors of the same length. Then `table(x,y)` will build a table with counts for each unique pair of occurrences in `x` and `y`.
- Some other functions you may find useful for the challenge: `colMeans`, `factor`, `head`, `length`, `max`, `read.csv`, `subset`, `tail`, `tapply`, `unique`, `which` and `which.max`.
- Record your solutions *as well as the code you used to obtain such solutions*.

Other topics

Why R?

When it comes to analyzing data, there are two competing paradigms. One could use point-and-click software with a graphical user interface, such as Excel, to perform calculations and draw graphs; or one could write programs that can be run to perform the analysis of the data, the generation of tables and statistics, and the production of figures automatically.

This latter approach is preferred because it allows for the *automation of analysis*, it requires a good documentation of the procedures, and is *completely replicable.*

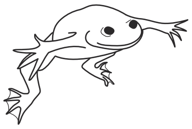
A few motivating examples:

- You have written code to analyze your data. You receive from your collaborators a new batch of data. With simple modifications of your code, you can update your results, tables and figures automatically.
- A new student joins the laboratory. The new student can read the code and understand the analysis without the need of a labmate showing the procedure step-by-step.
- The reviewers of your manuscript ask you to slightly alter the analysis. Rather than having to start over, you can modify a few lines of code and satisfy the reviewers.

R is a statistical software that is *completely programmable.* This means that one can write a program ("script") containing a series of commands for the analysis of data, and execute them automatically. This approach is especially good as it makes the analysis of data well-documented, and completely replicable.

R is *free software*: anyone can download its source code, modify it, and improve it. The R community of users is vast and very active. In particular, scientists have enthusiastically embraced the program, creating thousands of packages to perform specific types of analysis, and adding many new capabilities. See www.r-pkg.org for a listing of official packages (which have been vetted by R core developers); many more are available on Bioconductor, GitHub, and other websites.

R was modeled after the commercial statistical software S by Robert Gentleman and Ross Ihaka. The project was started in 1992, first released in 1994, and the first stable version appeared in 2000. Today, R is managed by the *R Core Team*.



X. laevis

Data subviews

When you are working with a much larger data set you need a strategy to inspect manageable subsets ("slices") of the data.

```
laptops <- read.csv("laptops.csv",comment = "#",stringsAsFactors = FALSE)
```

Each of the examples below print a subset of the data.

```
laptops[,2]
laptops[[2]]
laptops[,"os"]
laptops["os"]
laptops$os
laptops[4,]
laptops[4,2]
laptops[4,"os"]
laptops$os[4]
laptops[4,]$os
```

Here are a few slightly more complex examples:

```
laptops[1:4,]
laptops[,2:3]
laptops[,c("os","mem")]
laptops[c("os","mem")]
```

Once you are comfortable with the basic elements of selecting subsets, you can combine these elements in an endless variety of ways, e.g.,

```
laptops[c(1:3,5:7),c("name","os")]
```

Creating new data sets from subviews

The result of almost any calculation in R can be saved to an object. This includes selecting subsets. For example,

```
x <- laptops[1:10,]
print(x)
class(x)
```

creates a new data frame from the first 10 rows.

Conditional subviews

One powerful way to inspect subsets is by condition. For example, to view all the laptops with 4 processors, do

```
rows <- which(laptops$cpus == 4)
laptops[rows,]
```

Exercise: How would you select the subset of laptops that are running the Windows operating system?

Manipulating data

Up until now, we have treated the laptops data frame as if it were a static object. But like almost any other object in R, a data frame can be modified and overwritten. This is very powerful but also obviously dangerous! Here we will illustrate a few of the many kinds of data manipulations we can make, building on the elements we learned above. But before doing so, it is good practice to create a copy of the data frame in case we make a mistake along the way:

```r
laptops2 <- laptops
```

Note that `laptops` and `laptops2` are two *entirely independent* copies of the data. Although they are identical at first, as soon you a make change to `laptop2`, you will have two different data sets. Again, when you have multiple versions of a data set present in your environment, it is your responsibility to keep your environment organized.

You can overwrite individual entries of the data frame, or several entries at once:

```r
laptops2[c(1,2),"mem"] <- 2
```

Using a conditional subview, we can replace all instances of "mac" values in the "os" column with "macOS":

```r
rows <- which(laptops2$os == "mac")
laptops2[rows,"os"] <- "macOS"
```

If you don't like the column names, you can also change them, too:

```r
colnames(laptops2) <- c("name","OS","GB","CPUs")
```

You can reorder the rows or columns:

```r
rows <- order(laptops2$name)
laptops2 <- laptops2[rows,]
laptops2 <- laptops2[c("name","CPUs","GB","OS")]
```

Or you can even create new columns:

```r
laptops2$MB <- 1000 * laptops2$GB
```

P. polytes

Building a data frame

Above, we deconstructed the laptops data frame. We can also go in the reverse direction and build a data frame by joining together data vectors. Here's an illustration.

```
x <- laptops$os
y <- laptops$cpu
```

Given text data `x` and numeric data `y`, construct a new data frame:

```
dat <- data.frame(os = x,cpu = y)
```

Creating data

R also has many facilities for creating data structures from scratch.

The most basic tool is the `c` function, short for "combine". It can combine multiple objects or values:

```
x      <- c(2,3,5,27,31,13,17,19)
sex    <- c("M","M","F","M","F")            # Sex of Drosophila
weight <- c(0.230,0.281,0.228,0.260,0.231)  # Weight (in mg)
```

You can quickly generate sequences of numbers using `seq`. For example, this generates all odd numbers from 1 to 100:

```
x <- seq(from = 1,to = 100,by = 2)
```

For simpler number sequences, use the colon operator:

```
x <- 1:10
```

To repeat a value (or values), use `rep`:

```
x <- rep("treated",5)   # Treatment status
x <- rep(c(1,2,3),4)
```

Finally, there are many functions for generating random numbers. For example,

```
x <- runif(100)
```

Matrices

A matrix is like a data frame—it also has rows and columns. A key difference is that all the columns must be of the same type. Here's an example of a 2 x 2 matrix:

```
A <- matrix(c(1,2,3,4),2,2)  # Inputs are values, nrows, ncols.
```

In the case of numeric values, you can perform the usual operations on matrices (product, inverse, decomposition, *etc*).

```
A %*% A         # Matrix product
solve(A)        # Matrix inverse
t(A)            # Transpose
A %*% solve(A) # This should return the identity matrix.
```

Determine the dimensions of a matrix:

```
nrow(A)
ncol(A)
dim(A)
```

Use indices to access a particular row or column of a matrix:

```
Z <- matrix(1:9,3,3)
Z[1,]            # First row.
Z[,2]            # Second column.
Z[1:2,2:3]       # Submatrix with coefficients in rows 1 & 2, and columns 2 & 3.
Z[c(1,3),c(1,3)] # Indexing non-adjacent rows and columns.
```

Some operations apply to all elements of the matrix:

```
sum(Z)
mean(Z)
```

Question: When is it better to store data in a matrix instead of a data frame?

Arrays

If you need tables with more than two dimensions, use arrays:

```
A <- array(1:24,c(4,3,2))
```

A matrix is a special case of an array with two dimensions.

You can still determine the dimensions with `dim`:

```
dim(A)
```

And you can access the elements as for matrices. One thing to be careful about: R drops dimensions that are not needed. So, if you access a “slice” of a 3-dimensional array, you should obtain a matrix:

```
A[,2,]
dim(A[,2,])
class(A[,2,])
```

Lists

You have already worked with a list object (perhaps without realizing it). The laptops data frame is also a list:

```
is.list(laptops)
```

A data frame is a special kind of list—it is a list in which every list item is a vector of the same length. Lists also allow vectors of different lengths, for example, here we add an additional item (“date”) that has a length of 1:

```
mylist <- as.list(laptops)
mylist$last_modified <- "September 9, 2019"
```

We could not do this in a data frame.

A list is a very general and very widely used R data structure for storing complex data.

“`<-`” versus “=”

In R, a common point of confusion in is that the equality symbol (=) can also be used for creating or overwriting objects. Many people prefer to use =, but we recommend against using it because:

1. = is easily confused with ==. (How are = and == different?)
2. The = is also used for named arguments to functions, *which is different from assignment.*

Missing data

Finally, R allows most types of data—text, numeric, factors, *etc*—to take on a special value, `NA`. This is short for “not available” or “not assigned”, and is commonly called “missing data”. One special feature of R is that it often gracefully handles data sets containing missing data.

Exercise: Set a few entries in the `laptops` data frame to the value `NA`; for example, to set the first row and column to `NA` you could do `laptops[1,1] <- NA`. (You might want to make these changes to a copy of the data frame instead.) Then run `summary(laptops)`. How are the missing data reported in the summary?

M. musculus

Optional activity: Analyzing genetic data

In this activity, you will apply the tools we have developed so far to look at an example data set from human genetics: genotype data on chromosome 6 collected from European individuals. (Data adapted from the Human Genome Diversity Project by John Novembre.)

```
chr6 <- read.table("H938_Euro_chr6.geno",header = TRUE)
```

Setting `header = TRUE` means that we want to take the first line to be a header containing the column names. How big is this table?

```
nrow(chr6)
ncol(chr6)
```

It contains 7 columns and over 40,000 rows!

The table reports the number of homozygotes (`nA1A1`, `nA2A2`) and heterozygotes (`nA1A2`), for 43,141 single nucleotide polymorphisms (SNPs) obtained by sequencing European individuals. The other columns are:

- `CHR`: The chromosome (in this case, 6).
- `SNP`: The identifier of the Single Nucleotide Polymorphism.
- `A1`: One of the observed alleles.
- `A2`: The other allele.

Write R code to answer the following questions:

1. How many individuals were sampled? Find the maximum of the sum `nA1A1 + nA1A2 + nA2A2`. *Hint:* Recall that you can access the columns by index (e.g., `chr6[,5]`), or by name (e.g., `chr6$nA1A1`, or `chr6[,"nA1A1"]`).
2. Use the `rowSums` function to obtain the same answer.
3. How many SNPs have no heterozygotes (*i.e.*, no "A1A2")?
4. How many SNPs have less than 1% heterozygotes?

M. mulatta

Basic Computing 2 — Introduction to R, Part 2[*]

Peter Carbonetto and Aarti Venkat *University of Chicago*

Like Part 1, in Part 2 we will continue learning R by analyzing tabular data—that is, a *data frame*—but in Part 2 the table is much larger. Some of the skills we learned in Part 1 will transfer to this large-scale data setting. But we will also have to learn some new skills and practices to be able to cope with larger and more complex data. In Part 2, we will also sharpen our practices to prepare ourselves for analyzing data in the Real World.

The Tornado Super Outbreak of 1974

From *University of Chicago Magazine*, Fall, 2020:

> Fujita published his proposed tornado scale in 1971, but it needed a high-profile event to take root. On April 3, 1974, a tornado touched down in Morris, Illinois, around noon. Over the next 17 hours, 148 confirmed tornadoes tore through 13 states and Ontario, Canada. Following the 1974 Super Outbreak—one of the worst tornado outbreaks on record—Fujita and his team took a whirlwind airplane tour of more than 10,000 miles, surveying the ruins.

Fujita's scale is now known the "F-scale", and it scores tornadoes from F0 to F5 based on wind speeds and ensuing damage.

Analysis aims

Our main analysis aim is to uncover evidence for the 1974 Tornado Super Outbreak in data from NOAA's Severe Weather Data Inventory (SWDI). We will also perform a *comparative analysis* of these data to better understand the importance of this Super Outbreak in tornado climatology.

Do I have what I need?

From GitHub, download the tutorial materials to your computer, and make sure you know where to find them.

Obviously, you will need R, and, optionally, RStudio. Also, you will need the **ggplot2** and **cowplot** packages. If you haven't already done so, go ahead and install these two packages.

Where am I?

Make sure your R working directory in is the same directory containing the tutorial materials; you can run `getwd()` and `list.files()` to check this. (If you need to change your R working

[*] This document is included as part of the Basic Computing 2—Introduction to R tutorial packet for the BSD qBio Bootcamp, University of Chicago, 2022. **Current version**: August 11, 2022; **Corresponding author**: pcarbo@uchicago.edu. Thanks to Stefano Allesina, John Novembre, Stephanie Palmer and Matthew Stephens for their guidance.

directory, you can either use the `setwd()` function or, in RStudio, you can select **Session > Set Working Directory > Choose Directory. . . .**)

Quit applications that are not needed and other "clutter" to reduce distractions. *Remember, the computer is device of distraction.*

Also, it is best if you start with a fresh workspace; you can refresh your environment by selecting **Session > Clear Workspace** from the RStudio menu.

Initial steps to explore the SWDI data

Here we will reuse some of the skills we learned in Part 1 to get a basic understanding of the SWDI data.

Since the SWDI data are stored as a CSV file, you should now know what to do to import the data into R:

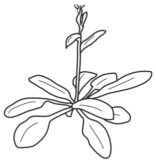
```
storms <- read.csv("StormEvents_details-ftp_v1.0_d1974_c20220425.csv.gz",
                   stringsAsFactors = FALSE)
```

Let's run a few lines of code to get an overview of the data frame:

Write your code here.

In our search for the Tornado Super Outbreak, we will use these seven columns: EVENT_TYPE, BEGIN_DAY, MONTH_NAME, STATE, BEGIN_LON, BEGIN_LAT and TOR_F_SCALE. Let's look more closely at these columns:

Write your code here.



A. thaliana

Preparing the tornado data

Our initial examinations suggest a few improvements to the data frame. *What are the improvements should we make?*

Write your notes here.

Now let's make these improvements:

Write your code here.

Since our focus is a particular type of storm event—tornadoes—let's extract the rows of the table that are relevant to our analysis.

Write your code here.

M. musculus

Keeping track of what we have done

Although we haven't made any breakthroughs yet, we have nonetheless made some progress: we have gained some basic insight into the data, and we have carefully prepared the data for our subsequent analyses. This is a good time to create a *script*—that is, the lines of code that reproduce the steps of our analysis. A script acts as both *a record of what we have done* and should, ideally, be *self-contained* so that it allow us to quickly *automate the analysis*.

Let's write a *minimal script* that reproduces the two most essential steps: (1) importing the data, and (2) prepping the data.

Write your code here.

Exercise: Check that your script is *reproducible* by running your script after clearing your workspace.

What are some reasons for writing a script?

Write your notes here.

Now that we have extracted and prepared the data, we are now ready to dive more deeply into these data. We will proceed by writing code to answer some questions.

When did the tornadoes occur?

Write your code to answer this question:

Write your code here.

Where did the tornadoes occur?

Write your code to answer this question. To help you with this, I have written a *custom function* that takes a data frame "latlongs" as input and outputs a map of the US with the geographic locations projected onto it:

```r
# Creates a map of the United States overlaid with points given by
# their geographic co-ordinates (lats and longs).
# The inputs are two numeric vectors of the same length.
# The output is a ggplot object.
map_usa_latlongs <- function (lats, longs) {
  dat <- data.frame(lat = lats,long = longs)
  return(ggplot(dat,aes(x = long,y = lat)) +
         geom_path(data = map_data("state"),
                   aes(x = long,y = lat,group = group),
                   color = "gray") +
         geom_point(shape = 20,size = 1) +
         theme_classic())
}
```

Write your code here.

What are the outliers?

Plotting the tornadoes by geographic location revealed some "outliers". Write some code to track down these outliers in the table, then remove them from the table:

Write your code here.

Exercise: Mapping the Super Outbreak

Create a map of the tornadoes again, this time focussed on the tornadoes that occurred on the single day of the tornado Super Outbreak. *You may be able to reuse the* `map_usa_latlongs` *custom function to answer this question.*

Write your code here.

Compare your map to https://en.wikipedia.org/wiki/1974_Super_Outbreak.

Activity: How unusual was the 1974 Tornado Super Outbreak?

Using our R skills, and the data provided by the NOAA, we recreated a detailed picture of the 1974 Tornado Super Outbreak—the same events that were carefully studied by Fujita and his colleagues.

Uncovering an exciting result—whether it is expected or surprising—is often only the beginning of your research. Typically, more analyses will need be done to confirm your result. This may involve collecting more data, or refining your analyses, or both.

As mentioned, creating *scripts* is an important element of your data analysis practice.

Here, we expand on this idea and develop a more flexible *generic script* that can be quickly reused to perform multiple analyses. As we develop more complex analyses, another important skill is knowing *when*—and *how*—to *reuse code*.

A natural question is whether there was something special about 1974. Are these "bursts" of tornadoes common, or exceedingly rare? To answer this question, we would need to download and analyze the storm event data from many years, not just 1974. Suddenly our large-scale data analysis has gotten even larger.

This is your task. Create a more flexible script that automates all the steps of the analysis: (1) load the data, (2) prepare the data, (3) extract the tornado data, and (4) create a plot showing number of tornadoes by date. This is mostly the same as what we did before, *but with a twist*: this new script is more flexible because it can analyze tornado event data from *any year*, not just 1974. To do this, define a new variable "year" and use this variable in your script to modify the analysis according to the target year.

Hint: You should be able to write this script by adapting the code we have written above. The function `paste` or `paste0` may also be useful.

The git repository includes SWDI data from 1974, 1975 and 1976 which you can use to test your script.

Optional: Download more storm event data from the NOAA SWDI database and try running your script on those data.

Discuss: Does this script achieve our goal of comparing tornado patterns across different years? What are ways in which this comparison could be improved?

Challenge problem: Use a "for loop" to automate your analysis—including the step to download the CSV files from the SWDI database—for all years from 1974 to 2020.

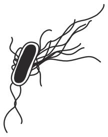
A few closing remarks

You will often find that it isn't long into a data analysis—sometimes after writing as little as 10–20 lines of code—when you run into problems. Some of these problems can be avoided with Good Practices. Examples of Good Practices include:

1. Keeping track of what you have done in *self-contained scripts.*
2. Giving names to variables that help you remember what they are for.
3. Including short, high-level comments explaining what the code does.

(How well did follow these practices in the examples above?)

One powerful idea is to identify a useful bit of code, then package this bit of code into a *function* that can be quickly reused.



E. coli

Programming challenge

Instructions

You will work with your group to solve the exercises below. When you have found the solutions, go to https://jnovembre.github.io/BSD-QBio8 and follow the link “Submit solution to challenge 2” to submit your answer.

Google Flu Trends

Google Flu started strong, with a paper in *Nature* (Ginsberg *et al*, 2009, doi:10.1038/nature07634) showing that, using data on Web search engine queries, one could predict the number of physician visits for influenza-like symptoms. Over time, the quality of predictions degraded considerably, requiring many adjustments to the model. Now defunct, Google Flu Trends has been proposed as a poster child of “Big Data hubris” (Lanzer *et al*, *Science*, 2014, doi:10.1126/science.1248506). In the folder containing the Basic Computing 2 tutorial materials, you will find the data used by Preis and Moat in their 2014 paper (doi:10.1098/rsos.140095) to show that, after accounting for some additional historical data, Google Flu Trends are correlated with outpatient visits due to influenza-like illnesses.

1. Read the data using function `read.csv` and plot the number of weekly outpatient visits versus the Google Flu Trends estimates.
2. Calculate the (Pearson’s) correlation using the `cor` function.
3. The data span 2010–2013. In August 2013, Google Flu changed their algorithm. Did this lead to improvements? Compare the data from weeks starting in August and September 2013 with the data from weeks starting in August and September 2010, 2011 and 2012. For each, calculate the correlation, and see whether the correlation is higher for 2013.

Hint: You will need to extract the year from a string for each row. There are several ways to do this, but one simple way is using `substr(gf$WeekCommencing,1,4)`, in which `gf` is the data frame containing the Google Flu data.

P. polytes

Other topics

Packages

R is the most popular statistical computing software among biologists. One reason for its popularity is the availability of many packages for tackling specialized research problems. These packages are often written by biologists for biologists. You can contribute a package, too! The RStudio website (goo.gl/harVqF) provides guidance on how to start developing R packages. See also Hadley Wickham's free online book (r-pkgs.had.co.nz).

You can often find highly specialized packages to address your research questions. Here are some suggestions for finding an appropriate package. The Comprehensive R Archive Network (CRAN) offers several ways to find specific packages for your task. You can browse the full list of CRAN packages (goo.gl/7oVyKC). Or you can go to the CRAN Task Views (goo.gl/0WdIcu) and browse a compilation of packages related to a topic or discipline.

From within R or RStudio, you can also call the function RSiteSearch("keyword"), which submits a search query to the website search.r-project.org. The website rseek.org casts an even wider net, as it not only includes package names and their documentation, but also blogs and mailing lists related to R. If your research interests are to high-throughput genomic data or other topics in bioinformatics or computational biology, you should also search the packages provided by Bioconductor (goo.gl/7dwQ1q).

Installing a package

Suppose you want to install the rsvd package. The **rsvd** package provides functions to quickly perform singular value decompositions (SVD) and principal components analysis (PCA) on large data sets. To install the package, run:

```
install.packages("rsvd")
```

Or, in RStudio, select the **Packages** panel, and click **Install**.

Loading packages

Once it is successfully installed, to load the **rsvd** package into your R environment, run:

```
library(rsvd)
```

Once you have loaded the package, you can use, for example, the rpca function for running PCA. To access the documentation that explains what rpca does, and how to use it, type

```
help(rpca)
```

Now suppose you would like to access the "bacteria" data set, which reports the incidence of *H. influenzae* in Australian children. The data set is included with **MASS** package. If you try to access these data before loading the package, you will get an error:

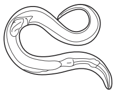
```
data(bacteria)
```

First, you need to load the package:

```
library(MASS)
```

Now the data set is available, and you can load it:

```
data(bacteria)
head(bacteria)
```



C. elegans

Random numbers

Your will sometimes need to generate random numbers. (They are actually "pseudorandom" numbers because they are not perfectly random.) In fact, random numbers are needed in the "case study" below.

`R` has many functions to sample random numbers from different statistical distributions. For example, use `runif` to create a vector containing 10 random numbers:

```
runif(10)
```

Question: What kind of random numbers are generated by `runif`? How could you check this?

To sample from a set of values, use `sample`:

```
v <- c("a", "b", "c", "d")
sample(v, 2)                # Sample without replacement.
sample(v, 6, replace = TRUE)  # Sample with replacement.
sample(v)                   # Shuffle the elements.
```

The normal distribution is one of the most commonly used distributions, so naturally there is a function in `R` for simulating from the normal distribution:

```
rnorm(3)                  # Three draws from the standard normal.
rnorm(3, mean = 5, sd = 4) # Change the mean and standard deviation.
```

Exercise: The normal distribution has a familiar shape. Use `rnorm` to generate a large number of values from the standard normal, then use `hist` to draw a histogram of these values (you can adjust the number of bins in the histogram with the `n` argument). Is the histogram "bell shaped"? Use `mean`, `median` and `sd` to verify that the random numbers recover the expected properties of the normal distribution. Here is some code to get you started:

```
set.seed(1)
x <- rnorm(10000)
hist(x, n = 64)
```

Why is `set.seed` useful? What happens if we remove the call to `set.seed`?

Writing functions

It is good practice to subdivide your analysis into functions, and then write a short "master" program that calls the functions and performs the analysis. In this way, the code will be more legible, easier to debug, and you will be able to recycle the functions for your other projects.

In `R`, every function has this form:

```
my_function_name <- function (arg1, arg2, arg3) {
  #
  # Body of the function.
  #
  return(return_value) # Not required, but most functions output something.
}
```

Here is a very simple example:

```
sum_two_numbers <- function (a, b) {
  s <- a + b
  return(s)
}
sum_two_numbers(5, 7.2)
```

In R, a function can return only one object. If you need to return multiple values, organize them into a vector, matrix or list, and return that; e.g.,

```
sum_and_prod <- function (a, b) {
  s <- a + b
  p <- a * b
  return(c(s,p))
}
sum_and_prod(5, 7.2)
```

Here is a more interesting function. It accepts two arguments, `x` and `s`, and returns the density of the normal distribution with zero mean and standard deviation `s` at `x`. This is the mathematical formula for the normal density with mean zero and standard deviation `s`:

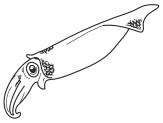
$$\frac{e^{-(x/s)^2/2}}{\sigma\sqrt{2\pi}}$$

Let's call this function `normpdf`:

```
normpdf <- function (x, s) {
  y <- exp(-(x/s)^2/2)/(sqrt(2*pi)*s)
  return(y)
}
```

Exercise: Check that this function gives the correct answers by comparing to the built-in function `dnorm`.

When developing and testing your function, remember this rule: *Whatever happens in the function stays in the function.* Inside `normpdf`, a new object, `y`, is created. But you will not see `y` in your environment (unless you happen to already have an object named `y`).



D. pealeii

Vectorization

`R` has a feature called *vectorization*—many operations in `R`, including most of the basic mathematical operations, are automatically applied to all values in a vector. (We briefly learned about vectorization in Basic Computing 1.) Let's check whether `R` automatically vectorizes the `normpdf` function by running this code:

```
n <- 1000
x <- seq(-3, 3, length.out = n)
y <- normpdf(x, 1)
print(y)
plot(x, y, type = "l")
```

Activity: Which operations in `normpdf` were applied 1,000 times, and which were applied just once? It may not be immediately obvious just by looking at the code, so to investigate this question, try running parts of the code in the console, e.g., `exp(-(x/s)^2/2)`, and see what happens.

Vectorization is very powerful, but it may take time to get comfortable with it, and know when it will work.

Conditional branching

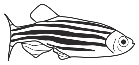
When we want a block of code to be executed only when a certain condition is met, we can write a conditional branching point. The syntax is as follows:

```
if (condition is met) {
  # Execute this block of code.
} else {
  # Execute this other block of code.
}
```

For example, try running these lines of code (you might want to try running them a few times):

```r
x <- rnorm(1)
if (x < 0) {
  msg <- paste(x, "is less than zero")
} else if (x > 0) {
  msg <- paste(x, "is greater than zero")
} else {
  msg <- paste(x, "is equal to zero")
}
print(msg)
```

We have created a conditional branching point, so that the value of `msg` changes depending on whether `x` is less than zero, greater than zero, or equal to zero.



D. rerio

Activity: Improved normal probability density function

The probability density function of the normal distribution is not defined if the standard devation is less than zero. When the standard deviation is exactly zero, the density is a “spike” at zero; it is `Inf` exactly at $x = 0$, and zero everywhere else. The pseudocode for this improved normal probability density function might look something like this:

```
normpdf(x, s)
  if s < 0
    return NaN
  else if s = 0
    if x = 0
      return Inf
    else
      return 0
  else
    evaluate normal pdf with s.d. s at x
```

Using this pseudocode as a guide, write an improved `normpdf` function:

Activity: The quadratic formula

There is a famous formula used to solve for x in the quadratic equation $ax^2 + bx + c = 0$. It is called the *quadratic formula.* Write down the quadratic formula here:

Write a function, `solvequad`, that takes three numbers as input (a, b and c) and returns the solution(s) x that are real (*i.e.*, not complex). **Hint:** Recall that a quadratic equation may have more than one real solution—or it may have none! You will need to use `if` and `else` to handle all possible cases.

Before writing any R code, first describe your `solvequad` function without worrying about R syntax—that is, using pseudocode:

Guided by your pseudocode, write the R code for your `solvequad` function:

After creating `solvequad`, check that it does the right thing by running these tests:

```
solvequad(4, 4, 1)    # Should return -1/2.
solvequad(1, -1, -2)  # Should return 2 and -1.
solvequad(1, 1, 1)    # Should return no solutions.
```

Run a few more tests using `www.wolframalpha.com`.

D. melanogaster

Looping

Another way to change the flow of your program is to write a loop. A loop is simply a series of commands that are repeated a number of times. For example, you want to run the same analysis on different data sets that you collected; you want to plot the results contained in a set of files; you want to test your simulation over a number of parameter sets; etc.

R provides you with two ways to loop over code blocks: the for loop and the while loop. Let's start with the for loop, which is used to iterate over a vector or list; for each value of the vector (or list), a series of commands will be run, as shown by the following example:

```r
v <- 1:10
for (i in v) {
  a <- i ^ 2
  print(a)
}
```

In the code above, the variable i takes the value of each element of v in sequence. Inside the block within the for loop, you can use the variable i to perform operations.

The anatomy of the for statement:

```r
for (variable in list_or_vector) {
  execute these commands
} # Automatically moves to the next value.
```

You should use a for loop when you know that you want to perform the analysis over a given set of values (e.g., files of a directory, rows in your data frames, sequences of a fasta file, *etc*).

The while loop is used when the commands need to be repeated while a certain condition is true, as shown by the following example:

```r
i <- 1
while (i <= 10) {
  a <- i ^ 2
  i <- i + 1
  print(a)
}
```

The script gives exactly the same result as the for loop above. A key difference is that you need to include a step to update the value of i, (using i <- i + 1), whereas in the for loop it is done for you automatically. The anatomy of the while statement:

```
while (condition is met) {
  execute these commands
} # Beware of infinite loops... remember to update the condition!
```

You can break a loop using `break`. For example:

```
i <- 1
while (TRUE) {
  if (i > 10) {
    break
  }
  a <- i ^ 2
  i <- i + 1
  print(a)
}
```

Question: Above, we ran three different loops, we found that each of them accomplished the same thing. Is one approach better? Why?

Creating computational notebooks using R Markdown

> *Let us change our traditional attitude to the construction of programs: instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to humans what we want the computer to do.*
>
> Donald E. Knuth, *Literate Programming*, 1984

When doing experiments, it is important to develop the habit of writing down everything you do in a laboratory notebook. That way, when writing your manuscript, responding to queries or discussing progress with your advisor, you can go back to your notes to find exactly what you did, how you did it, and possibly *why* you did it. The same should be true for computational work.

`RStudio` makes it very easy to build a computational laboratory notebook. First, create a new `R Markdown` file. (Choose **File > New File > R Markdown** from the RStudio menu bar.)

An R Markdown file is simply a text file. But it is interpreted in a special way that allows the text to be transformed it into a webpage (`.html`) or PDF file. You can use special syntax to render the text in different ways. Here are a few examples of R Markdown syntax:

```
*Italic text* **Bold text**

# Very large header

## Large header

### Smaller header

Unordered and ordered lists:
```

```
+ First
+ Second
    + Second 1
    + Second 2

1. This is a
2. Numbered list
```

When rendered as a PDF, the above R Markdown looks like this:

Very large header

Large header

Smaller header

Italic text **Bold text**

Unordered and ordered lists:

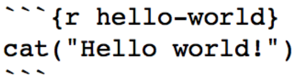
- First
- Second
    - Second 1
    - Second 2

1. This is a
2. Numbered list

You can also insert inline code by enclosing it in backticks.

The most important feature of R Markdown is that you can include blocks of code, and they will be interpreted and executed by R. You can therefore combine effectively the code itself with the description of what you are doing.

For example, including a code chunk in your R Markdown file,



~~~
```{r hello-world}
cat("Hello world!")
```
~~~

will render a document containing both the results and the code that run to generate those results:

```
cat("Hello world!")
# Hello world!
```

If you don't want to run the R code, but just display it, use {r hello-world, eval = FALSE}; if you want to show the output but not the code, use {r hello-world, echo = FALSE}.

You can include plots, tables, and even mathematical equations using LaTeX. In summary, when exploring your data, or describing the methods for your paper, give R Markdown a try!

You can find inspiration in this Boot Camp; the materials for Basic and Advanced Computing were written in R Markdown.

M. mulatta

Bonus case study: Do shorter titles lead to more citations?

To keep learning about R, we study the following question:

Is the length of a paper title related to the number of citations?

This is what Letchford *et al* claimed (doi:10.1098/rsos.150266). In 2015, they analyzed 140,000 papers, and they found that *shorter titles* were associated with a larger number of citations.

In the folder containing the Basic Computing 2 tutorial materials, you will find data on scientific articles published between 2004 and 2013 in three top disciplinary journals, *Nature Neuroscience*, *Nature Genetics* and *Ecology Letters*. These data are conatined in three CSV files. We are going to use these data to explore this question.

Load data

Start by reading in the data:

```
data.file <- file.path("citations", "nature_neuroscience.csv")
papers    <- read.csv(data.file, stringsAsFactors = FALSE)
```

Next, take a peek at the data. How large is it?

```
nrow(papers)
ncol(papers)
```

Let's see the first few rows:

```
head(papers)
```

The goal is to test whether papers with longer titles accrue fewer (or perhaps more?) citations than those with shorter titles. The first step is to add another column to the data containing the length of the title for each paper:

```
papers$TitleLength <- nchar(papers$Title)
```

Basic statistics

In the original paper, Letchford *et al* used rank-correlation: rank all the papers according to their title length and the number of citations. If Kendall's τ ("rank correlation") is positive, then longer titles are associated with *more* citations; if τ is negative, longer titles are associated with *fewer* citations. In R, you can compute rank correlation using cor:

```
k <- cor(papers$TitleLength, papers$Cited.by, method = "kendall")
```

To perform a significance test for correlation, use cor.test:

```
k.test <- cor.test(papers$TitleLength, papers$Cited.by, method = "kendall")
```

Does the output of cor.test show that the correlation between the ranks is positive or negative? Is this positive or negative correlation significant? You should find that the correlation is opposite of the one reported by Letchford *et al*—longer titles are associated with *more* citations!

Now we are going to examine the data in a different way to test whether this result is robust.

Basic plotting

To plot title length vs. number of citations, we need to learn about plotting in R. To produce a simple scatterplot using the base plotting functions, simply run:

```
plot(papers$TitleLength, papers$Cited.by)
```

The problem with this very simple plot is that it is hard to detect any trend—a few papers have many more citations than the rest, obscuring the data at the bottom of the plot. This suggests that plotting the data on the *logarithmic scale* is a better approach:

```
plot(papers$TitleLength, log10(papers$Cited.by))
```

Question: This is a better plot, but there is one problem with it. What is the problem, and what fix would you suggest? Write your code to fix the problem:

Again, it is hard to see any trend in here. Maybe we should plot the best fitting line and overlay it on top of the graph. To do so, we first need to learn about linear regressions in R.

Linear regression

R was born for statistics — the fact that it is very easy to fit a linear regression in is not surprising! To build a linear model comparing columns x and y in data frame, dat, use lm, the "Swiss army knife" for linear regression in R:

```
model <- lm(y ~ x, dat)  # y = a + b*x + error
```

Let's perform a linear regression of the number of citations (on the log-scale) vs. title length. To do so, first create a new column in the data frame containing the counts on the log-scale:

```
papers$LogCits <- log10(papers$Cited.by + 1)
```

Now you can perform a linear regression:

```
model_citations <- lm(LogCits ~ TitleLength, papers)
model_citations           # Gives best-fit line.
summary(model_citations)  # Gives more info.
```

And we can easily add this best-fit line to our plot:

```
plot(papers$TitleLength, papers$LogCits)
abline(model_citations, col = "red", lty = "dotted")
```

Once we add the best-fit line, the positive trend is more clear.

One thing to consider is that this data set spans a decade. Naturally, older papers have had more time to accrue citations. In our models, we should control for this effect. But first, we should explore whether this is an important factor to consider.

First, let's plot the distribution of the number of citations for all the papers:

```
hist(papers$LogCits)
```

You can control the number of histogram bins with the n argument:

```
hist(papers$LogCits, n = 50)
```

Alternatively, estimate the density using density, then plot it:

```
plot(density(papers$LogCits))
```

Next, compare the distributions for papers published in 2004, 2009 and 2013:

```
plot(density(papers$LogCits[papers$Year == 2004]), col = "black")
lines(density(papers$LogCits[papers$Year == 2009]), col = "blue")
lines(density(papers$LogCits[papers$Year == 2013]), col = "red")
```

More recent papers should have fewer citations. Does your plot support this hypothesis? You can account for this in your regression model by incorporating the year of publication into the linear regression:

```r
model_citations_better <- lm(LogCits ~ TitleLength + Year, papers)
summary(model_citations_better)
```

Does the regression coefficient (slope) for year confirm that older papers have more citations?

Our new analysis is better than before, but might be even better to have a separate "baseline" for each year. This can be done by converting the "Year" column to a factor:

```r
papers$Year            <- factor(papers$Year)
model_citations_better <- lm(LogCits ~ Year + TitleLength, papers)
summary(model_citations_better)
```

This model has a different baseline for each year, and then title length influences this baseline. In this new model, are longer titles still associated with more citations?

Computing *p*-values using randomization

Kendall's τ takes as input two rankings, x and y, both of the same length, n. It calculates the number of "concordant pairs" (if $x_i > x_j$, then $y_i > y_j$) and the number of "discordant pairs". The final value is

$$\tau = \frac{n_{\text{concordant}} - n_{\text{discordant}}}{\frac{n(n-1)}{2}}$$

If x and y are completely independent, we would expect τ to have a distribution centered at zero. The variance of the "null" distribution of τ depends on the data. It is typically approximated as a normal distribution. If you want to have a stronger result that does not rely on a normality assumption, you can use randomizations to calculate a p-value. Simply, compute τ for the actual data, as well as for many "fake" datasets obtained by randomizing the data. Your p-value is then the proportion of τ values for the randomized sets that exceed the τ value for the actual data.

Here, we will try to implement this randomization to calculate a p-value for papers published in 2006, and then we will compare against the p-value obtained from running `cor.test`. To do this, we will use a for-loop for the randomization.

First, subset the data:

```r
dat <- papers[papers$Year == 2006, ]
```

Compute τ from these data:

```r
k <- cor(dat$TitleLength, dat$Cited.by, method = "kendall")
```

Now calculate τ in "fake" data sets by randomly scrambling the citation counts. Begin by doing this for one fake data set:

```r
shuffled_citation_counts <- sample(dat$Cited.by)
k.fake <- cor(dat$TitleLength, shuffled_citation_counts, method = "kendall")
```

Is the value of τ closer to zero in this "shuffled" data set?

To get an accurate p-value, we should compute τ for a large number of shuffled data sets. Let's try 1,000 of them. This and similar randomization techniques are known as "bootstrapping".

```r
nr     <- 1000         # Number of fake data sets.
k.fake <- rep(0, nr)  # Storage all the "fake" taus.
```

Since this computation involves lots of repetition, a for-loop makes a lot of sense here:

```r
for (i in 1:nr){
  shuffled_citation_counts <- sample(dat$Cited.by)
  k.fake[i] <- cor(dat$TitleLength, shuffled_citation_counts,
                   method = "kendall")
}
```

After running this loop, you should have 1,000 correlations calculated from 1,000 fake data sets. You have just generated a "null" distribution for τ. What does this null distribution look like? Try plotting it:

```r
hist(k.fake, n = 50)
```

What proportion of the fake data sets have a correlation that exceeds the correlation in the actual data? This is the p-value.

```r
pvalue <- mean(k.fake >= k)
```

How does your new p-value compare to the p-value computed by `cor.test`? Is it smaller or larger?

```r
cor.test(dat$TitleLength, dat$Cited.by, method = "kendall")
```

Question: Did you get the same result as the instructor, or your neighbours? If not, why? How could you ensure that your result is more similar, or the same?

Whenever possible, use randomizations rather than relying on classical tests. They are more difficult to implement, and more computationally expensive, but they allow you to avoid making assumptions about your data.

Repeating the analysis for each year

Up until this point, we have only analyzed the citation data for 2006. Does the result we obtained for 2006 hold up in other years? Let's explore this question—we will use a for-loop to repeat the analysis for 2004 to 2013. Let's be smart about designing our code and use a *function* to decompose the problem into parts. The code for the final analysis will look like this:

```
years <- 2004:2013
for (i in years){
  dat <- papers[papers$Year == i, ]
  out <- analyze_citations(dat)
  cat("year:", i, "tau:", out$k, "pvalue:", out$pvalue, "\n")
}
```

The missing piece is the code implementing function `analyze_citations`. You can re-use your code above to write this function.

```
analyze_citations <- function (dat) {
  nr     <- 1000
  k      <- cor(dat$TitleLength, dat$Cited.by, method = "kendall")
  k.fake <- rep(0, nr)
  for (i in 1:nr) {
    k.fake[i] <- cor(dat$TitleLength, sample(dat$Cited.by), method = "kendall")
  }
  return(list(k = k, pvalue = mean(k.fake >= k)))
}
```

Activity: Organizing and running your code

Now we would like to be able to automate the analysis, such that we can repeat it for each journal. This is a good place to pause and introduce how to go about writing programs that are well-organized, easy to write, easy to debug, and easier to reuse.

1. Take the problem, and divide it into smaller tasks (these are the functions).
2. Write the code for each task (function) separately, and make sure it does what it is supposed to do.
3. Document the code so that you can later understand what you did, how you did it, and why.
4. Combine the functions into a master program.

For example, let's say we want to write a program that takes as input the name of files containing citation data. The program should first fit a linear regression model,

```
log(citations + 1) ~ as.factor(Year) + TitleLength
```

then output the coefficient associated with `TitleLength`, and its *p*-value.

We could split the program into the following tasks:

1. A function to load and prepare the data for a linear regression analysis.
2. A function to run the linear regression analysis.
3. A master code that puts it all together.

Let's begin with the master code—the bulk of the code is a for-loop that repeats the regression analysis for each journal:

```
files <- list.files("citations", full.names = TRUE)
for (i in files) {
  cat("Analyzing data from",i,"\n")
  papers <- load_citation_data(i)
  out    <- fit_citation_model(papers)
  cat("coefficient:", out$estimate, "p-value:", out$pvalue, "\n")
}
```

This code doesn't work yet because you haven't written the functions that are called inside the loop. (What error message to you get when you try to run the code?)

The `load_citation_data` function reads in the data from the CSV file, then prepares the data for the linear regression analysis:

```
load_citation_data <- function (filename) {
  dat <- read.csv(filename, stringsAsFactors = FALSE)
  dat$TitleLength <- nchar(dat$Title)
  dat$LogCits     <- log10(dat$Cited.by + 1)
  dat$Year        <- as.factor(dat$Year)
  return(dat)
}
```

Before continuing, check that it works by running it on one of the CSV files:

```
papers <- load_citation_data("citations/nature_neuroscience.csv")
```

The `fit_citation_model` function fits a linear regression model to the input data, then extracts the quantities from the regression analysis we are most interested in (the "best-fit" slope and the *p*-value corresponding to "TitleLength").

```
fit_citation_model <- function (papers) {
  model <- lm(LogCits ~ Year + TitleLength, papers)
  terms <- summary(model)$coefficients
  return(list(estimate = terms["TitleLength", "Estimate"],
              pvalue   = terms["TitleLength", "Pr(>|t|)"]))
}
```

Check that this function runs, and does what it is supposed to do:

```
out <- fit_citation_model(papers)
```

Now that you have defined the necessary functions, try running the master code above.

Question: Suppose you download a fourth CSV file containing data on papers from the *American Journal of Human Genetics*. Would any changes need to be made to your R code above to run it on the four citation data sets?

Advanced Computing — Data wrangling and plotting*

Stefano Allesina *University of Chicago*

Data wrangling

As biologists living in the XXI century, we are often faced with tons of data, possibly replicated over several organisms, treatments, or locations. We would like to streamline and automate our analysis as much as possible, writing scripts that are easy to read, fast to run, and easy to debug. Base `R` can get the job done, but often the code contains complicated operations (think of the cases in which you used `lapply` only because of its speed), and a lot of `$` signs and brackets.

To start, we need to import `tidyverse`:

```r
library(tidyverse)
```

`tidyverse` is a fantastic bundle of packages: a collection of `R` packages designed to manipulate large data frames in a simple and straightforward way. These tools are also much faster than the corresponding base `R` commands, and allow you to write compact code by concatenating commands to build "pipelines". Moreover, all of the packages in the bundle share the same philosophy, and are seamlessly integrated. By default, calling `library(tidyverse)` loads the packages `readr`, `tidyr` and `dplyr` (to read, organize and manipulate data), `ggplot2` (data plotting), `stringr` (string manipulation) and a few others; many others ancillary packages that are part of the `tidyverse` can be loaded if needed.

Then, we need a dataset to play with. We take a dataset containing all the papers published by UofC researchers in *Nature* or *Science* between 1999 and July 2019:

```r
pubs <- read.csv("../data/UC_Nat_Sci_1999-2019.csv")
```

A new data type, `tibble`

The data are stored in a `data.frame`:

```r
is.data.frame(pubs)
```

`tidyverse` ships with a new data type, called a `tibble`. It also comes with its improved function to read data:

```r
pubs <- read_csv("../data/UC_Nat_Sci_1999-2019.csv")
pubs
```

which automatically reads the data as a `tibble`. The nice feature of `tibble` objects is that they will print only what fits on the screen, and also give you useful information on the size of the data,

*This document is included as part of the Advanced Computing tutorial packet for the BSD qBio Bootcamp, University of Chicago, 2021. **Current version**: August 26, 2021.

as well as the type of data in each column. Other than that, a tibble object behaves very much like a data.frame. If you want to transform the tibble back into a data.frame, use the function as.data.frame(my_tibble); the function as_tibble(my_data_frame) transforms a data.frame into a tibble.

We can take a look at the data using one of several functions:

- head(pubs) shows the first few rows
- tail(pubs) shows the last few rows
- glimpse(pubs) a summary of the data (similar to str in base R)
- View(pubs) open data in spreadsheet-like window

Selecting rows and columns

There are many ways to subset the data, either by row (subsetting the *observations*), or by column (subsetting the *variables*). For example, let's select only articles published after 2009:

```
filter(pubs, Year > 2009)
```

You can see that 515 of the 953 documents were published in the last 10 years. We have used the command filter(tbl, conditions) to select certain observations. We can combine several conditions, by listing them side by side, possibly using logical operators.

> **Exercise:** what does this do?
>
> ```
> filter(pubs, Year == 2008, ‘Source title‘ == "Nature", ‘Cited by‘ > 100)
> ```

Note that the "back ticks" can be used to type column names that contain spaces and non-standard characters. This is nice, because otherwise the name of the column would need to be altered (as done automatically by read.csv, sometimes creating column names that are difficult to interpret or type).

We can also select particular variables using the function select(tbl, cols to select). For example, select only Authors and Title:

```
select(pubs, Authors, Title)
```

How many years are represented in the data set? We can use the function distinct(tbl) to retain only the rows that differ from each other:

```
distinct(select(pubs, Year))
```

Where we first extracted only the column Year, and then retained only distinct values.

Other ways to subset observations:

- sample_n(tbl, howmany, replace = TRUE) sample howmany rows at random with replacement

- `sample_frac(tbl, proportion, replace = FALSE)` sample a certain proportion (e.g. `0.2` for 20%) of rows at random without replacement
- `slice(tbl, 50:100)` extract the rows between 50 and 100
- `top_n(tbl, 10, Year)` extract the first 10 rows, once ordered by `Year`

More ways to select columns:

- `select(pubs, contains("Cited"))` select all columns containing the word `Cited`
- `select(pubs, -Authors, -Year)` exclude the columns `Authors` and `Year`
- `select(pubs, matches("astring|anotherstring"))` select all columns whose names match a regular expression.

Creating pipelines using %>%

We've been calling nested functions, such as `distinct(select(pubs, ...))`. If you have to add another layer or two, the code would become unreadable. `dplyr` allows you to "un-nest" these functions and create a "pipeline", in which you concatenate commands separated by the special operator %>%. For example:

```
pubs %>% # take a data table
  select(Year) %>% # select a columns
  distinct() # remove duplicates
```

does exactly the same as the command we've run above, but is much more readable. By concatenating many commands, you can create incredibly complex pipelines while retaining readability.

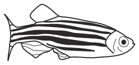
Producing summaries

Sometimes we need to calculate statistics on certain columns. For example, calculate the average number of citations. We can do this using `summarise`:

```
pubs %>% summarise(avg = mean(`Cited by`))
```

which returns a `tibble` object with just the average number of citations. You can combine multiple statistics (use `first`, `last`, `min`, `max`, `n` [count the number of rows], `n_distinct` [count the number of distinct rows], `mean`, `median`, `var`, `sd`, etc.):

```
pubs %>% summarise(avg = mean(`Cited by`),
                   sd = sd(`Cited by`),
                   median = median(`Cited by`))
```



D. rerio

Summaries by group

One of the most useful features of `dplyr` is the ability to produce statistics for the data once subsetted by *groups*. For example, we would like to compute the average number of citations by journal and year:

```
pubs %>%
  group_by(‘Source title‘, Year) %>%
  summarise(avg = mean(‘Cited by‘))
```

> **Exercise:** count the number of articles by UofC researcher in *Nature* and *Science* by `Source title` and `Year`.

Ordering the data

To order the data according to one or more variables, use `arrange()`:

```
pubs %>% select(Title, ‘Cited by‘) %>% arrange(‘Cited by‘)
pubs %>% select(Title, ‘Cited by‘) %>% arrange(desc(‘Cited by‘))
```

Renaming columns

To rename one or more columns, use `rename()`:

```
pubs %>% rename(Cites = ‘Cited by‘)
```

If you want to retain the new name(s), simply overwrite the object:

```
pubs <- pubs %>% rename(Cites = ‘Cited by‘, Journal = ‘Source title‘)
```

Adding new variables using `mutate`

If you want to add one or more new columns, use the function `mutate`. For example, suppose we want to count the number of authors for each document. Authors are separated by commas (with small errors, but let's disregard that), and therefore a strategy would be to first count the number of commas, and then add 1:

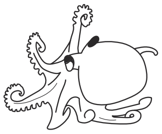
```
pubs <- pubs %>% mutate(Num_authors = str_count(Authors, ",") + 1)
```

use the function `transmute()` to create a new column and drop the original columns. You can also use `mutate` and `transmute` on grouped data.

When writing code, it is good practice to separate the operations by line:

```
# A more complex example: for each paper,
# compute the percentile rank of citations
# compared to other papers of the same year
pubs %>%
  group_by(Year) %>% # group papers according to year
  mutate(pr = percent_rank(Cites)) %>% # compute % rank by Citations
  ungroup() %>% # remove group information
  arrange(Year, desc(pr), Authors) %>% # order by Year then % rank (decreasing)
  head(20) # display first 20 rows
```

in this way, you can easily comment out a part of the pipeline (or add another piece in the middle).



O. bimaculoides

Data plotting

The most salient feature of scientific graphs should be clarity. Each figure should make crystal-clear a) what is being plotted; b) what are the axes; c) what do colors, shapes, and sizes represent; d) the message the figure wants to convey. Each figure is accompanied by a (sometimes long) caption, where the details can be explained further, but the main message should be clear from glancing at the figure (often, figures are the first thing editors and referees look at).

Many scientific publications contain very poor graphics: labels are missing, scales are unintelligible, there is no explanation of some graphical elements. Moreover, some color graphs are impossible to understand if printed in black and white, or difficult to discern for color-blind people (8% of men, 0.5% of women).

Given the effort that you put in your science, you want to ensure that it is well presented and accessible. The investment to master some plotting software will be rewarded by pleasing graphics that convey a clear message.

In this section, we introduce `ggplot2`, a plotting package for `R` This package was developed by Hadley Wickham who contributed many important packages to `R` (including `dplyr`), and who is the force behing `tidyverse`. Unlike many other plotting systems, `ggplot2` is deeply rooted in a "philosophical" vision. The goal is to conceive a grammar for all graphical representation of data. Leland Wilkinson and collaborators proposed The Grammar of Graphics. It follows the idea of a well-formed sentence that is composed of a subject, a predicate, and an object. The Grammar of Graphics likewise aims at describing a well-formed graph by a grammar that captures a very wide range of statistical and scientific graphics. This might be more clear with an example – Take a simple two-dimensional scatterplot. How can we describe it? We have:

- **Data** The data we want to plot.
- **Mapping** What part of the data is associated with a particular visual feature? For example: Which column is associated with the x-axis? Which with the y-axis? Which column corresponds to the shape or the color of the points? In `ggplot2` lingo, these are called *aesthetic*

mappings (aes).

- **Geometry** Do we want to draw points? Lines? In ggplot2 we speak of *geometries* (geom).
- **Scale** Do we want the sizes and shapes of the points to scale according to some value? Linearly? Logarithmically? Which palette of colors do we want to use?
- **Coordinate** We need to choose a coordinate system (e.g., Cartesian, polar).
- **Faceting** Do we want to produce different panels, partitioning the data according to one (or more) of the variables?

This basic grammar can be extended by adding statistical transformations of the data (e.g., regression, smoothing), multiple layers, adjustment of position (e.g., stack bars instead of plotting them side-by-side), annotations, and so on.

Exactly like in the grammar of a natural language, we can easily change the meaning of a "sentence" by adding or removing parts. Also, it is very easy to completely change the type of geometry if we are moving from say a histogram to a boxplot or a violin plot, as these types of plots are meant to describe one-dimensional distributions. Similarly, we can go from points to lines, changing one "word" in our code. Finally, the look and feel of the graphs is controlled by a theming system, separating the content from the presentation.

Basic ggplot2

ggplot2 ships with a simplified graphing function, called qplot. In this introduction we are not going to use it, and we concentrate instead on the function ggplot, which gives you complete control over your plotting. First, we need to load the package (note that ggplot2 is automatically loaded by tidyverse). While we are at it, let's also load a package extending its theming system:

```r
library(ggplot2)
library(ggthemes)
```

A particularity of ggplot2 is that it accepts exclusively data organized in tables (a data.frame or a tibble object). Thus, all of your data needs to be converted into a table format for plotting.

For our first plot, we're going to produce a barplot showing the number of papers in Science and Nature by UofC researcher for each Year. To start:

```r
ggplot(data = pubs)
```

As you can see, nothing is drawn: we need to specify what we would like to associate to the *x* axis (i.e., we want to set the *aesthetic mappings*):

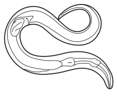
```r
ggplot(data = pubs) + aes(x = Year)
```

Note that we concatenate pieces of our "sentence" using the + sign! We've got the axes, but still no graph… we need to specify a geometry. Let's use barplot:

```r
ggplot(data = pubs) + aes(x = Year) + geom_bar()
```

As you can see, we wrote a well-formed sentence, composed of **data** + **mapping** + **geometry**, and this has produced a well-formed plot. We can add other mappings, for example, showing the journal in which the paper was published:

```r
ggplot(data = pubs) + aes(x = Year, fill = Journal) + geom_bar()
```



C. elegans

Scatterplots

Using ggplot2, one can produce very many types of graphs. The package works very well for 2D graphs (or 3D rendered in two dimensions), while it lack capabilities to draw proper 3D graphs, or networks.

The main feature of ggplot2 is that you can tinker with your graph fairly easily, and with a common grammar. You don't have to settle on a certain presentation of the data until you're ready, and it is very easy to switch from one type of graph to another.

For example, let's plot the number of citations in the y axis, the year in the x axis. We want a scatterplot, which is produced by the geometry geom_point:

```r
pl <- ggplot(data = pubs) + # data
  aes(x = Year, y = Cites) + # aesthetic mappings
  geom_point() # geometry

pl # or show(pl)
```

This does not look very good, because some papers have a much larger number of citations than other. We can attempt plotting the log(Cites + 1) instead (the +1 is added because some papers might have 0 citations):

```r
pl <- ggplot(data = pubs) + # data
  aes(x = Year, y = log(Cites + 1)) + # aesthetic mappings
  geom_point() # geometry

pl # or show(pl)
```

Much nicer! Now we can add a smoother by typing:

```r
pl + geom_smooth() # spline by default
pl + geom_smooth(method = "lm", se = FALSE) # linear model, no standard errors
```

> **Exercise:** repeat the plot of the citations, but showing a different colour for each journal; add a smoother for each journal separately. Do papers receive more citations when they're published in *Nature* or *Science*?

Histograms, density and boxplots

What is the distribution of citations?

```r
ggplot(data = pubs) + aes(x = Cites) + geom_histogram()
```

You can see that there are some papers with many more citations than others. Try log-transforming the data:

```r
ggplot(data = pubs) + aes(x = log(Cites + 1)) + geom_histogram()
```

Now we observe an histogram much closer to a Normal distribution, meaning that the number of citations is approximately log-normally distributed. You can switch to a density plot quite easily (just change the geometry!):

```r
ggplot(data = pubs) + aes(x = log(Cites + 1)) + geom_density()
```

Similarly, we can produce boxplots, for example showing the number of citations for papers in *Nature* and *Science* (log transformed):

```r
ggplot(data = pubs) + aes(x = Journal, y = log(Cites + 1)) + geom_boxplot()
```

It is very easy to change geometry, for example switching to a violin plot:

```r
ggplot(data = pubs) + aes(x = Journal, y = log(Cites + 1)) + geom_violin()
```

Exercise:

- Produce a boxplot showing the number of authors (in log) per year (use `factor(Year)` for the x axis). Is science becoming more collaborative?
- Now produce a scatterplot showing the same trend, and add a smoothing function.

Scales

We can use scales to determine how the aesthetic mappings are displayed. For example, we could set the x axis to be in logarithmic scale, or we can choose how the colors, shapes and sizes are used. `ggplot2` uses two types of scales: `continuous` scales are used for continuos variables (e.g., real numbers); `discrete` scales for variables that can only take a certain number of values (e.g., treatments, labels, factors, etc.).

For example, let's plot a histogram showing the number of authors per paper:

```r
ggplot(pubs, aes(x = Num_authors)) + geom_histogram() # no transformation
ggplot(pubs, aes(x = Num_authors)) + geom_histogram() +
  scale_x_continuous(trans = "log") # natural log
```

```
ggplot(pubs, aes(x = Num_authors)) + geom_histogram() +
  scale_x_continuous(trans = "log10") # base 10 log
ggplot(pubs, aes(x = Num_authors)) + geom_histogram() +
  scale_x_continuous(trans = "sqrt", name = "Number of authors")
ggplot(pubs, aes(x = Num_authors)) + geom_histogram() + scale_x_log10() # shorthand
```

We can use different color scales. For example:

```
pl <- ggplot(data = pubs %>% filter(Year %in% c(2000, 2005, 2010, 2015))) +
  aes(x = Num_authors, y = Cites, colour = factor(Year)) +
  geom_point() +
  scale_x_log10() +
  scale_y_log10()
pl + scale_colour_brewer()
pl + scale_colour_brewer(palette = "Spectral")
pl + scale_colour_brewer(palette = "Set1")
pl + scale_colour_brewer("year of publication", palette = "Paired")
```

Or use the number of authors a continuous variable:

```
pl <- ggplot(data = pubs) +
  aes(x = Year, y = log(Cites + 1), colour = log(Num_authors)) +
  geom_point()
pl + scale_colour_gradient()
pl + scale_colour_gradient(low = "red", high = "green")
pl + scale_colour_gradientn(colours = c("blue", "white", "red"))
```

Similarly, you can use scales to modify the display of the shapes of the points (`scale_shape_contintuous`, `scale_shape_discrete`), their size (`scale_size_continuous`, `scale_size_discrete`), etc. To set values manually (useful typically for discrete scales of colors or shapes), use `scale_colour_manual`, `scale_shape_manual` etc.

Themes

Themes allow you to manipulate the look and feel of a graph with just one command. The package `ggthemes` extends the themes collection of `ggplot2` considerably. For example:

```
library(ggthemes)
pl + theme_bw() # white background
pl + theme_economist() # like in the magazine "The Economist"
pl + theme_wsj() # like "The Wall Street Journal"
```

Faceting

In many cases, we would like to produce a multi-panel graph, in which each panel shows the data for a certain combination of parameters. In `ggplot` this is called *faceting*: the command `facet_grid` is used when you want to produce a grid of panels, in which all the panels in the same row (column) have axis-ranges in common; `facet_wrap` is used when the different panels do not have axis-ranges in common.

For example:

```r
pl <- ggplot(data = pubs %>% filter(Year %in% c(2000, 2005, 2010, 2015))) +
  aes(x = log10(Cites + 1)) +
  geom_histogram()
show(pl)
pl + facet_grid(~Year) # in the same row
pl + facet_grid(Year~.) # col
pl + facet_grid(Journal ~ Year) # two facet variables
pl + facet_wrap(Journal ~ Year, scales = "free") # just wrap around
```

Setting features

Often, you want to simply set a feature (e.g., the color of the points, or their shape), rather than using it to display information (i.e., mapping some aestethic). In such cases, simply declare the feature outside the `aes`:

```r
pl <- ggplot(data = pubs %>% filter(Year %in% c(2000, 2005, 2010, 2015))) +
  aes(x = log10(Num_authors))
pl + geom_histogram()
pl + geom_histogram(colour = "red", fill = "lightblue")
```

Saving graphs

You can either save graphs as done normally in `R`:

```r
# save to pdf format
pdf("my_output.pdf", width = 6, height = 4)
print(my_plot)
dev.off()
# save to svg format
svg("my_output.svg", width = 6, height = 4)
print(my_plot)
dev.off()
```

or use the function ggsave

```r
# save current graph
ggsave("my_output.pdf")
# save a graph stored in ggplot object
ggsave(plot = my_plot, filename = "my_output.svg")
```

Multiple layers

Finally, you can overlay different data sets, using different geometries. For example, suppose that we have two data sets: one for papers with few authors (say <10) and one for large collaborations:

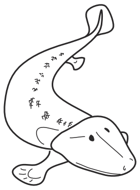
```
small_collab <- pubs %>% filter(Num_authors < 10)
large_collab <- pubs %>% filter(Num_authors >= 10)
```

We can overlay different geometries for the same data set:

```
ggplot(data = small_collab) +
  aes(x = factor(Num_authors), y = log(Cites + 1)) +
  geom_boxplot(fill = "lightblue") +
  geom_violin(fill = "NA") +
  geom_point(alpha = 0.25) # alpha stands for transparency
```

Or combine different data sets (with the same `aes`!):

```
ggplot(data = small_collab) +
  aes(x = Year) +
  geom_bar(fill = "red", alpha = 0.5) +
  geom_bar(data = large_collab, fill = "blue", alpha = 0.5)
```



T. roseae

Tidying up data

The best way to organize data for plotting and computing is the *tidy form*, meaning that a) each variable has its own column, and b) each observation has its own row. When data are not in tidy form, you can use the package `tidyr` to reshape them.

For example, suppose we want to produce a table in which for each journal and year, we report the average number of authors. First, we need to compute the values:

```
avg_authors <- pubs %>%
  group_by(Journal, Year) %>%
  summarise(avg_au = mean(Num_authors))
```

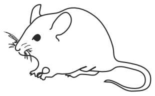
This table is in tidy format (also called “narrow” format); we want to create columns for each journal, and report the average in the corresponding cell. To do so, we “spread” the journals into columns:

```
avg_authors <- avg_authors %>% spread(Journal, avg_au)
```

Note that this is not in tidy form, as two observations are in the same row (also called "messy" or "wide" format). While this is not ideal for computing, it is great for human consumption, as we can easily compare the two numbers in the same row.

If we want to go back to tidy form, we can "gather" the column names, and return to tidy:

```
# gather(where to store col names,
#        where to store values,
#        which columns to gather)
avg_authors %>% gather(Journal, Average_num_authors, 2:3)
# alternatively, if it's cleaner
avg_authors %>% gather(Journal, Average_num_authors, -Year)
```



M. musculus

Joining tables

If you have multiple data frames or `tibble` objects with shared columns, it is easy to join them (as in a database). To showcase this, we are going to extract papers by very prolific authors. First, we want to compute how many papers are in the data for each "author" (actually, last-name initial combinations, which might represent different authors with common names…). First, we need a data set in which the authors have been separated:

```
by_author <- pubs %>%
  select(Authors, Title) %>%
  separate_rows(sep = ", ", Authors) %>%
  rename(Focal_author = Authors)
```

Now we can count the number of appearances of each name:

```
by_author <- by_author %>%
  group_by(Focal_author) %>%
  mutate(Tot = n())
```

Where we have created a new column (`Tot`) by calling `mutate` on grouped data. Who are the authors most represented in the data?

```
tot_author <- by_author %>%
  select(Focal_author, Tot) %>%
  distinct() %>%
  arrange(desc(Tot))
```

You can see that common Chinese name combinations are in the top few rows (meaning that probably we conflated several authors. . .). Let's plot an histogram:

```r
tot_author %>% ggplot() + aes(x = Tot) + geom_histogram() + scale_y_log10()
```

As you can see, the vast majority of authors appears only once, and very few, appear 15 or more times. We want to extract the papers of the most prolific authors from the data that we have stored in `pubs`. For example, we want to consider authors that are represented 10 or more times in these papers. To do so, we first extract the prolific authors:

```r
prolific <- by_author %>% filter(Tot >= 10)
```

and now we can join `pubs` and `prolific`. By calling `inner_join`, only rows that are present in both tables will be retained; because the two tables share a column (`Title`), `dplyr` can proceed automatically:

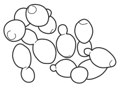
```r
pubs %>% inner_join(prolific)
```

We can use this table to compute the number of citations received by each prolific author:

```r
pubs %>% inner_join(prolific) %>% ggplot() +
  aes(x = Focal_author, y = Cites) +
  geom_col() + # similar to bar plot
  theme(axis.text.x = element_text(angle = 90, hjust = 1)) # rotate labels
```

Besides `inner_join(x, y)`, you can use:

- `left_join(x, y)`: return all rows from `x`, and all columns from `x` and `y` (those with no match will show `NA`);
- `right_join(x. y)`: return all rows from `y`, and all columns from `x` and `y`;
- `full_join(x, y)`: return all rows and all columns from both `x` and `y`. Where there are not matching values, returns `NA` for the one missing;
- `anti_join(x, y)`: return all rows from `x` where there are not matching values in `y`.


S. cerevisiae

Data visualization tutorial: exploring data and telling stories using ggplot2[*]

Peter Carbonetto *University of Chicago*

In this lesson, we will learn how to use ggplot2 to create simple yet effective data visualizations. The ggplot2 package is an incredibly powerful plotting interface that extends the base plotting functions in R.

Some motivation

A good figure is an important part of an impactful research paper or presentation. A good figure is one that *tells an interesting story.*

Almost inevitably, creating a good figure takes iteration and refinement. You will rarely get it right on the first try.

For these reasons, taking the *programmatic approach* to creating plots is very powerful. It allows you to:

a. Create an endless variety of plots.
b. Reuse code to quickly create and revise plots.

In this tutorial we will explore the programmatic approach to plotting using **ggplot2**.

Setup

Do I have what I need? From GitHub, download the tutorial materials to your computer, and make sure you know where to find them.

Launch RStudio.

If you have not already done so, install these packages:

```
install.packages("ggplot2")
install.packages("cowplot")
install.packages("ggrepel")
install.packages("htmlwidgets")
install.packages("plotly")
```

Where am I? Make sure your R working directory in is the same directory containing the tutorial materials; you can run `getwd()` and `list.files()` to check this. (If you need to change your R

[*] This document is included as part of the Data Visualization tutorial packet for the BSD qBio8 Bootcamp, University of Chicago, 2022. **Current version**: August 11, 2022; **Corresponding author**: pcarbo@uchicago.edu. Thanks to John Novembre, Stephanie Palmer, Stefano Allesina and Matthew Stephens for their ideas and support.

working directory, you can either use the `setwd()` function or, in RStudio, you can select **Session > Set Working Directory > Choose Directory. . . .**)

Quit applications that are not needed and other "clutter" to reduce distractions. *Remember, the computer is device of distraction.*

Also, it is best if you start with a fresh workspace; you can refresh your environment by selecting **Session > Clear Workspace** from the RStudio menu.

Do smaller dogs live longer?

The study of dogs is a surprisingly fruitful area of research! In this tutorial, we will make use of some data that was made available by the authors of a 2008 *Genetics* article, *Single-nucleotide-polymorphism-based association mapping of dog stereotypes.* These data are stored in a CSV file.

Our main analysis aim is to investigate the anecdotal claim that smaller breeds (such as Chihuahuas) live longer than larger breeds (such as Saint Bernards).

Our first ggplot plot

It will take us some time to understand *how* ggplot works, but let's start by quickly creating our first ggplot.

```
library(ggplot2)
dogs <- read.csv("dogs.csv",stringsAsFactors = FALSE)
ggplot(dogs,aes_string(x = "height",y = "weight")) +
  geom_point()
```

✎ *Write your notes here.*

Let's now take a closer look at the data, and we will return to the ggplot code later.

A first look at the data

When you load data into R for the first time, it is important to get a basic understanding of the data frame and its contents.

Write your code here.

What different types of data are in this table?

Write your notes here.

The often overlooked scatterplot

In this tutorial, we will learn about ggplot2 through one of the most basic data visualizations: *the scatterplot.*

The scatterplot is easily overlooked because it is so simple. But it can be one of the most effective ways to visualize relationships. And it has many uses.

With embellishments (adding labels, varying color, shape, size, *etc.*), scatterplots can produce stunning visualizations.

Our first ggplot2 (with "ugly" code)

Recall, our objective is to investigate the relationship between size and longevity in dogs. We'll use weight as a proxy for size.

Write your code here.

Now for some more elegant code can accomplish the same thing. (This more elegant code comes with a "for experts only" warning.)

Write your code here.

For the moment I want to focus on the "uglier" code because it highlights better the key elements of a ggplot2 plot:

1. The first input is the data (stored in a data frame).
2. The second input is an "aesthetic mapping", created using `aes_string` that defines how columns are mapped to features of the plot (axes, shapes, colors, *etc.*). *Note:* Many people use `aes` instead of `aes_string`. The `aes` variant is more powerful, but can also be more difficult to use.
3. A "geom", short for "geometric object", specifies the type of plot. ggplot2 has an excellent on-line reference at ggplot2.tidyverse.org explaining all the "geoms", from bar charts to contour plots, with code examples for each.
4. ggplot2 outputs a *ggplot object* `p`, which can be drawn to the screen with `print(p)` or just `p` (then hit "enter" or "return").

The distinguishing feature of ggplot2 is that plots are created by *adding layers*. This layering allows for infinite variety of plots to be created. The layering approach means that ggplot2 is easily extendible, and many R packages have been developed to enhance ggplot2. (We will use two of these packages, ggrepel and cowplot.)

A few improvements

No plot is ever right the first time. *What are ways we can improve our plot?* Add code for your improved plot here:

Write your code here.

In ggplot, plots can be improved either by modifying the inputs to the functions you are already using (for example, `geom_point` has many settings that can be fiddled with), or by *adding layers*. The ggplot online reference has many helpful examples that illustrate the variety of ways plots can be improved.

D. melanogaster

Save your work

We've worked hard, and make considerable progress. This is a good point to save our work in an image file that can be shared with others. *What type of image file should we use?*

Write your code here.

Plot the best-fit line

It has been estimated that an increase of 28 lbs in a dog's body weight corresponds to about a 1-year drop in expected lifespan (with a maximum lifespan of about 13 years). How well does this estimate agree with our data? Let's investigate this question by plotting the line that *best fits* these data (specifically, this is the "least-squares" fit).

```
fit <- lm(aod ~ weight,dogs)
coef(fit)
```

Now let's add this "best fit" line to the plot. In case we make a mistake, let's call our new plot "p2" to avoid writing over our previous plot:

Write your code here.

Now let's add another another "abline" layer to compare our estimate against the previous estimate (and name the new plot object "p3"):

Write your code here.

Notice how easy it was to add layers to an existing plot!

Which breeds fit the trend, and which don't?

It would be helpful if we could tell which breeds are being plotted. Adding text labels to a ggplot is also done by adding a layer.

There's one catch here—there simply isn't enough real estate on the plot to accommodate all the breed names. This is a great opportunity to play around with a clever package, **ggrepel**, that adds the labels in a way that makes them more readable, and only adds them to the plot when possible. Let's appreciate how simply this sophisticated plot is created.

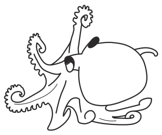
```
library(ggrepel)
p4 <- p3 + geom_text_repel(mapping = aes_string(label = "breed"),
                           size = 2.5,color = "gray")
```

Notice that the labels are automatically redrawn as the plot is resized. Give it a try!

A QTL for weight

In the *Genetics* paper, the strongest *quantitative trait locus* (QTL) for weight was a QTL on chromosome 7. Using your ggplot coding skills, create a scatterplot to visualize the relationship between weight and the allele frequency at this QTL. *Is your code reproducible? i.e., could I run your code to get the same plot?*

Write your code here.



O. bimaculoides

Exploration: a surprising subtlety with color

So far, we have focussed on using the co-ordinate plane to visualize relationships. But other "aesthetics"—color, size, shape, *etc.*—can also be used to tell an evocative story. In the last chapter of this tutorial, we will explore the use of color to visualize relationships. In so doing, we will discover a complication.

We learned that the "shortcoat" column had values of 0 or 1 (with a few NAs).

Let's try varying the color of the points using the shortcoat column:

```
p <- ggplot(dogs,aes_string(x = "weight",y = "aod",color = "shortcoat")) +
  geom_point() +
  theme_cowplot()
p
```

Are we happy with this plot? How could it be improved? Write your code for an improved plot here:

Write your code here.

Optional: Try varying the shape of the points instead of color. If you have have written your ggplot code well, this should only involve a slight change to your code. You can use `scale_shape_manual` to select the shapes. Running `plot(0:23,pch = 0:23)` will give you the full list of shapes to choose from.

Write your code here.

More on color

Carefully chosen colors can often be the difference between an effective visualization and an ineffective one. One the few complaints I have with ggplot2 is that the default colors choices are quite poor. So when you use ggplot2, you will often need to make adjustments to the colors. One rule-of-thumb is that "warmer" colors such as red and orange tend to draw the reader's attention.

There are several good resources on use of color in data visualization, and I will mention a couple here: Color Brewer (https://colorbrewer2.org); and a short article, "Color blindness", written by Bang Wong (*Nature Methods*, 2011). For more discussion, see the "Fundamentals of data visualization" book by Claus Wilke.

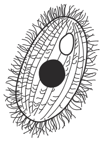
In our scatterplot, the best color choice is less clear, so I will let you experiment with different choices. To override the color defaults, add a `scale_color_manual` layer:

Write your code here.

There are several different ways to specify colors. I prefer specifying colors by name; to get the full list of color names, run `colors()`.

The function that controls the color of a discrete variable has an odd name: `scale_color_manual`. This is because, in ggplot2, all methods that control the mapping of variables to colors, shapes, sizes, axes, *etc.*, start with `scale_`.

Obviously, there is much more to ggplot2. But once you are comfortable with these basic elements, you will find that almost everything else in ggplot2 is a variation of what we covered in this lesson.



T. thermophila

Programming challenge: Mapping the genetic basis of physiological and behavioral traits in outbred mice

In this programming challenge, you will use simple visualizations to gain insight into biological data.

You are working in a lab studying the genetics of physiological and behavioral traits in mice. The lab has just completed a large study of mice from an outbred mouse population, "CFW" ("Carworth Farms White"). The aim of the study is to identify genetic contributors to variation in behaviour and musculoskeletal traits.

Note hese challenges are roughly ordered in increasing level of complexity. *Do not be discouraged if you have difficulty completing every one.*

Collaboration strategy

Before diving into the problems, first agree on a collaboration strategy with your teammates. Important aspects include communication and co-ordination practices, and setting goals and deadlines. How will your team collaborate on code, and share solutions? (Consider online resources such as Etherpad or the UofC-hosted Google Drive.) The aim is not just to tackle the challenges, but also to do so collaboratively.

Instructions

- Locate the files for this exercise on your computer (see "Materials" below).
- Make sure your R working directory is set to the same directory containing the tutorial materials; use `getwd()` to check this. (If you need to change your R working directory, you can either use the `setwd()` function or, in RStudio, you can select **Session > Set Working Directory > Choose Directory. . . .**)
- Some of the programming challenges require uploading an image file containing a plot. Use `ggsave` to save your plot as a file. Any standard image format (e.g., PDF, PNG) is fine.
- No additional R packages are needed beyond what we used in the examples above.

Materials

- **pheno.csv:** CSV file containing physiological and behavioral phenotype data on 1,219 male mice from the CFW outbred mouse stock. Data are from Parker *et al*, 2016. Use **readpheno.R** to read the phenotype data from the CSV file into a data frame. After filtering out some of the samples, this script should create a new data frame, `pheno`, containing phenotype data on 1,092 samples (these are the rows of the data frame).
- **hmdp.csv:** CSV file containing bone-mineral density measurements taken in 878 male mice from the Hybrid Mouse Diversity Panel (HMDP). Data are from Farber *et al*, 2011. To load the data into your R environment, run this code:

  ```
  hmdp <- read.csv("hmdp.csv",stringsAsFactors = FALSE)
  ```

  This will create a data frame, `hmdp`, containing BMD data on 878 mice (these are the rows of the data frame).

- **gwscan.csv:** CSV file containing results of a "genome-wide scan" for abnormal BMD. (The association p-values were computed using GEMMA 0.96.) To read the results of the genome-wide scan, run the following code:

```
gwscan <- read.csv("gwscan.csv",stringsAsFactors = FALSE)
gwscan <- transform(gwscan,chr = factor(chr,1:19))
```

  This will create a data frame, `gwscan`. Each row of the data frame is a genetic variant (a single nucleotide polymorphism, or "SNP"). The columns are chromosome ("chr"), base-pair position on the chromosome ("pos"), and the p-value for a test of association between variant genotype and trait value ("abnormalBMD"). The value stored in the "abnormalBMD" column is $-\log_{10}(P)$, where P is the p-value.
- **geno_rs29477109.csv:** CSV file containing estimated genotypes at one SNP (rs29477109) for 1,038 CFW mice. Use the following code to read the genotype data into your R environment:

```
geno <- read.csv("geno_rs29477109.csv",stringsAsFactors = FALSE)
geno <- transform(geno,id = as.character(id))
```

  This will create a new data frame, `geno`, with 1,038 rows (samples). The genotypes are encoded as "dosages"—that is, the expected number of times the alternative allele is observed in the genotype. This will be an integer (0, 1, 2), or a real number between 0 and 2 when there is some uncertainty in the estimate of the genotype. For this SNP, the reference allele is T and the alternative allele is C. Therefore, dosages 0, 1 and 2 correspond to genotypes TT, CT and CC, respectively (genotypes CT and TC are equivalent).
- **wtccc.png:** Example genome-wide scan ("Manhattan plot") from Fig. 4 of the WTCCC paper. The p-values highlighted in green show the regions of the human genome most strongly associated with Crohn's disease risk.

A couple tips

- Some "geoms" you may find useful: `geom_point`, `geom_histogram`, `geom_boxplot`.
- In some cases it may be useful to convert to a *factor*.

Part A: Exploratory analysis of muscle development and conditioned fear data

Your first task is to create plots to explore the data.

1. A basic initial step in an exploratory analysis is to visualize the distribution of the data. It is often convenient if the distribution is normal, or "bell shaped".
   - Visualize the empirical distribution of tibialis anterior (TA) muscle weight (column "TA") with a histogram. Units are mg. *Hint:* Try using function `geom_histogram`.
   - Is the distribution of TA weight roughly normal? Are there mice with unusually large or unusually small values ("outliers")? If so, how many "outliers" are there?
2. It is also important to understand relationships among measured quantities. For example, the development of the tibia bone (column "tibia") could influence TA muscle weight. Create a scatterplot (`geom_point`) to visualize the relationship between TA weight and tibia length. (Tibia length units are mm.) Based on this plot, what can you say about the relationship between TA weight and tibia length? Quantify this relationship by fitting a linear model,

before and after removing the outlying TA values. (*Hint*: Use the lm and summary functions. See also the "r.squared" return value in help(summary.lm).)

3. The "AvToneD3" column contains data collected from a behavioral test called the "Conditioned Fear" test.
   - Visualize the empirical distribution of AvToneD3 ("freezing to cue") with a histogram. Is the distribution of AvToneD3 approximately normal?
   - Freezing to cue is a proportion (a number between 0 and 1). A common way to obtain a more "normal" quantity is to transform it using the "logit" function.[1] Visualize the empirical distribution of the logit-transformed phenotype. Is the transformed phenotype more "bell shaped"? After the transformation, do you observe unusually small or unusually large values?
   - A common concern with behavioral tests is that the testing devices can lead to measurement error. It is especially a concern when multiple devices are used, as the devices can give slightly different measurements, even after careful calibration. Create a plot to visualize the relationship between (transformed) freezing to cue and the device used ("FCbox" column). *Hint:* Try a boxplot (geom_boxplot). Based on this plot, does the apparatus used affect these behavioral test measurements?

Part B: Exploratory analysis of bone-mineral density data

Now you will examine data on bone-mineral density (BMD) in mice. This is a trait that is important for studying human diseases such as osteoporosis (units are mg/cm^2).

- Plot the distribution of BMD in CFW mice (see column "BMD"). What is most notable about the distribution?
- Compare these data against BMD measurements taken in a "reference" mouse population, the Hybrid Mouse Diversity Panel. To compare, create two histograms, and draw one on top of the other. What difference do you observe in the BMD distributions? For a correct comparison, consider that: (1) BMD in CFW mice was measured in the femurs of male mice only; (2) BMD in HMDP mice was recorded in g/cm^2. *Hints:* Potentially useful functions include xlim and labs from the ggplot2 package, and plot_grid from the cowplot package. The binwidth argument in geom_histogram may also be useful.

Part C: Mapping the genetic basis of osteopetrotic bones

A binary trait, "abnormal BMD", was defined that signals whether an individual mouse had "abnormal", or osteopetrotic, bones. It takes a value of 1 when BMD falls on the "long tail" of the distribution (BMD greater than 90 mg/cm^2), otherwise zero.

GEMMA was used to carry out a "genome-wide association study" (GWAS) for this trait; that is, support for association with abnormal BMD was evaluated at 79,824 genetic variants (single nucleotide polymorphisms, or "SNPs") on chromosomes 1–19. At each SNP, a p-value quantifies the support for an association with abnormal BMD.

1. Your first task is to get an overview of the association results by creating a "Manhattan plot". Follow as closely as possible the provided prototype, **wtccc.png**, which shows a genome-

[1] R code: `logit <- function(x) log((x + 0.001) / (1 - x + 0.001))`

wide scan for Crohn's disease. (Don't worry about highlighting the strongest *p*-values in green.) *Hints:* Replicating some elements of this plot may be more challenging than others, so start with a simple plot, and try to improve on it. Recall the adage that creating plots requires relatively little effort *provided the data are in the right form*—consider adding appropriate columns to the `gwscan` data frame before writing your ggplot code. Functions from the ggplot2 package that you may find useful include `geom_point`, `scale_color_manual` and `scale_x_continuous`.

- In your plot, you should observe that the most strongly associated SNPs cluster closely together in small regions of the genome. This is common—it is due to a genetic phenomenon known as linkage disequilibrium (LD). It is a consequence of low recombination rates between markers in small populations. How many SNPs have "strong" statistical support for association with abnormal BMD, specifically with a $-\log_{10} p$-value > 6? How many "distinct" regions of the genome are strongly associated with abnormal BMD at this *p*-value threshold?
- What *p*-value does a $-\log_{10} p$-value of 6 correspond to?
- Using your plot, identify the "distinct region" (this is called a "quantitative trait locus", or QTL) with the strongest association signal. What is, roughly, the size of the QTL in Megabases (Mb) if we define the QTL by base-pair positions of the SNPs with $-\log_{10} p$-value > 6? Using the UCSC Genome Browser, get a rough count of the number of genes that are *transcribed* in this region. (Parker *et al*, 2016 identified *Col1a1* as a candidate BMD gene.) *Hint:* All SNP positions are based on *NCBI Mouse Genome Assembly 38* (mm10, December 2011).

2. Your next task is to visualize the relationship between genotype and phenotype. From the genome-wide scan of abnormal BMD, you should find that rs29477109 is the SNP most strongly associated with abnormal BMD. Here you will look closely at the relationship between BMD and the genotype at this SNP. In developing your visualization, consider that:

   - The samples listed in the phenotype and genotype tables are not the same. So you will need to align the two tables to properly show analyze the relationship. *Hint:* Function `match` could be useful for this.
   - The genotypes, stored in file **geno_rs29477109.csv**, are encoded as "dosages" (numbers between 0 and 2). You could start with a scatterplot of BMD vs. dosage. But ultimately it is more effective if the genotypes (CC, CT and TT) are plotted instead. *Hints:* In effect, what you need to do is convert from a continuous variable (dosage) to a discrete variable (genotype). One approach is to create a `factor` column from the "dosage" column. (For dosages that are not exactly 0, 1 or 2, you could simply round to the nearest whole number.) A boxplot is recommended; see function `geom_boxplot`.

   Based on your plot, how would describe (in plain language) the relationship between the genotype and BMD?

Notes

An interactive plot

Here is an example of an *interactive plot* created using plotly. There are some similarities between this code and our ggplot code above.

```
library(plotly)
library(htmlwidgets)
p <- plot_ly(data = dogs,type = "scatter",mode = "markers",
             x = ~weight,y = ~aod,
             text = ~sprintf("%s\n%0.1f lbs\n%0.1f years",breed,weight,aod),
             marker = list(line = list(color = "white",width = 1),size = 9),
             hoverinfo = "text",width = 550,height = 500)
p <- layout(p,
            hoverlabel = list(bgcolor = "white",bordercolor = "black"))
saveWidget(p,"dogs.html",selfcontained = TRUE)
```

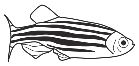
Useful resources

- ggplot2 reference
- Fundamentals of Data Visualization by Claus Wilke.

Data sources

- The dog breeds data were downloaded from the Genetics journal website.
- The CFW phenotype and genotype data were downloaded from Data Dryad.

License



D. rerio

Defensive Programming in R[*]

Sarah Cobey (*original author*) *University of Chicago*
Kyle Hernandez *University of Chicago*

Goal: Convince new and existing programmers of the importance of defensive programming practices, introduce general programming principles, and provide specific tips for programming and debugging in R. **Audience:** Scientific researchers who use R and believe the accuracy of their code and/or efficiency of their programming could be improved.

Installation

For people who have completed the other tutorials, there is nothing new to install. For others starting fresh, install R and RStudio. To install R, follow instructions at cran.rstudio.com. Then install Rstudio following the instructions at https://www.rstudio.com/products/rstudio/download/.

Motivation

Defensive programming is the practice of anticipating errors in your code and handling them efficiently.

If you're new to programming, defensive programming might seem tedious at first. But if you've been programming long, you've probably experienced firsthand the stress from

- inexplicable, strange behavior by the code
- code that seems to work under some conditions but not others
- incorrect results or bugs that take days or weeks to fix
- a program that seems to produce the correct results but then, months or years later, gives you an answer that you know must be wrong... thereby putting all previous results in doubt
- the nagging feeling that maybe there's still a bug somewhere
- not getting others' code to run or run correctly, even though you're following their instructions

Defensive programming is thus also a set of practices for preserving sanity and conducting research efficiently. It is an art, in that the best methods vary from person to person and from project to project. As you will see, which techniques you use depend on the kind of mistakes you make, who else will use your code, and the project's requirements for accuracy and robustness. But that flexibility does not imply defensive programming is "optional": steady scientific progress depends on it. In general, we need scientific code to be perfectly accurate (or at least have well understood inaccuracies), but compared to other programmers, we are less concerned with security and ensuring that completely naive users can run our programs under diverse circumstances (although standards here are changing).

[*] This document is included as part of the 'Defensive Programming in R' tutorial packet for the BSD qBio Bootcamp, University of Chicago, 2021. **Current version**: August 11, 2022.

In the first part of this tutorial, we will review key principles of defensive programming for scientific researchers. These principles hold for all languages, not just `R`. In the second part, we will consider R-specific debugging practices in more depth.

Part 1: Principles

Part 1 focuses on defense. You saw a few of these principles in Basic Computing 2, but they are important enough to be repeated here.

1. Before writing code, draft your program as a series of modular functions with high-level documentation, and develop tests for it.
2. Write clearly and cautiously.
3. Develop one function at a time, and test it before writing another.
4. Document often.
5. Refactor often.
6. When you run your code, save each "experiment" in a separate directory with a complete copy of the code repository and all parameters.
7. Don't be *too* defensive.

In part 2, we will focus on what to do when tests (from Principle 3) indicate something is wrong.

Principle 1. Before writing code, draft your program as a series of modular functions with high-level documentation, and develop tests for it.

Many of us have had the experience of writing a long paper only to realize, several pages in, that we in fact need to say something slightly different. Restructuring a paper at this stage can be a real chore. Outlining your code as you would outline a paper avoids this problem. In fact, outlining your code can be even more helpful because it helps you think not just generally about how your algorithm will flow and where you want to end up, but also about what building blocks (functions and containers) you'll use to get there. Thinking about the "big picture" design will prevent you from coding yourself into a tight spot—when you realize hundreds of lines in that you should've been tracking past states of a variable, for instance, but your current containers are only storing the current state. Drafting also makes the actual writing much more easy and fun.

For complex programs, your draft may start as a diagram or flowchart showing how major functions and variables relate to one another or brief notes about each of step of the algorithm. These outlines are known as pseudocode, and there are myriad customs for pseudocode and programmers loyal to particular styles. For simple scripts, it is often sufficient to write pseudocode as placeholder comments. For instance, if we are simulating a population in which individuals can be born or die (and nothing else happens), we could write:

```
# initialize population size (N), birth rate (b), death rate (d),
#            total simulation time (t_max), and start time (t=0)
# while N > 0 and t < t_max
# ... generate random numbers to determine whether next event is birth or death
#     and time to event (dt)
# ... update N (increment if birth, decrement if death)
# ... update time t to t+dt
```

Here, `b` and `d` are per capita rates. This is an example of the Gillespie algorithm. It was initially developed as an exact stochastic approach for simulating chemical reaction kinetics, and it is widely used in biology, chemistry, and physics.

Can you see some limitations of the pseudocode so far? First, it lacks obvious modularity, though this is partly due to its vagueness. The first step under the `while` loop could become its own function that is defined separately. Second, it is missing a critical feature, in that it's not obvious what is being output: do we want the population size at the end of the simulation, or the population size at each event? If the latter, we may need to initialize a container, such as a dataframe, in which we store the value of `N` and `t` at every event. After further thought, we might decide such a container would be too big—perhaps we only need to know the value of `N` at 1/1000th the typical rate of births, and so we might introduce an additional loop to store `N` only when the new time `(t+dt)` exceeds the most recent prescribed observation/sampling time. This sampling time would need to be stored in an extra variable, and we could also make the sampling procedure into its own function. And maybe we want a function to plot `N` over time at the end.

The next stage of drafting is to consider how the code might go wrong, and what it would take to convince ourselves that it is accurate. We'll spend more time on this later, but now is the time to think of every possible sanity check for the code. Sometimes this can lead to changes in code design.

> **Exercise**
>
> What sanity checks and tests would you include for the code above?

Some examples:

- Initial values of `b`, `d`, and `t_max` should be non-negative and not change
- The population size `N` should probably start as a positive whole number and never fall below zero
- If the birth and death rates are zero, `N` should not change
- If the birth rate equals the death rate, on average, `N` should not change when it is large
- The ratio of births to deaths should equal the ratio of the birth rate to the death rate, on average
- The population should, on average, increase exponentially at rate `b-d` (or decrease exponentially if `d>b`)

Some of these criteria arise from common sense or assumptions we want to build into the model (for instance, that the birth and death rates aren't negative), and others we know from the mathematics of the system. For instance, the population cannot change if there are no births and deaths, and it must increase on average if the birth rate exceeds the death rate. It helps that the Gillespie algorithm represents kinetics that can be written as simple differential equations. However, because the simulations are stochastic, we need to look at many realizations (randomizations, trajectories) to ensure there aren't consistent biases. Just as with "real" data, we must use statistics to confirm that the distribution of `N` in 10,000 simulations after 100 time units is not significantly different from what we would predict mathematically.

The bottom line is that we have identified some tests for the (1) inputs, (2) intermediate variable states (such as `N`), (3) final outputs to test that the program is running correctly. Note that one of our tests, the fraction of birth to death events, is not something we were originally tracking, and thus we might decide now to create a separate function and variables to handle these quantities. We will talk in more detail about how to implement these tests in Principle 3. Generally, tests of specific functions are known as **unit tests**, and tests of aggregate behavior (like the trajectories of 10,000 simulations) are known as **system tests**. As a general principle, we need to have both, and we need as much as possible to compare their outputs to analytic expectations. It is also very useful to identify what you want to see right away. For instance, you may want to write a function to plot the population size over time *before* you code anything else because having immediate visual feedback can be extremely helpful (and inspiring!).

> “Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a *computer* what to do, let us concentrate rather on explaining to *human beings* what we want a computer to do.” -Donald Knuth

This is also a good time to draft very high-level documentation for your code, for instance in a `readme.MD` file. What are the inputs, what does the code do, and what does it return?

> **Exercise**
>
> Assume there are two discrete populations. Each has nonoverlapping generations and the same generation time. Their per capita birth rates are `b1` and `b2`. Some of the newborns migrate between populations.
>
> - Write pseudocode to calculate the distribution of population frequencies after 100 generations.
> - Is your code optimally modular?
> - What are the inputs and outputs of the program? Of each function?
> - How could you test the inputs, functions, and overall program?
> - Discuss your approach with your neighbor.

Principle 2. Write clearly and cautiously.

You’re already on your way to writing clearly and cautiously if you outline your program before you start writing it. Here we’ll discuss some practices to follow as you write.

A general rule is that it is more important for scientific code to be readable and correct than it is for it to be fast. Do not obsess too much about the efficiency of the code when writing.

> “Premature optimization is the root of all evil.” -Donald Knuth

Develop useful conventions for your variable and function names. There are many conventions, some followed more religiously than others. It’s most important to be consistent within your own code.

- Don’t make yourself and others guess at meaning by making names too short. It’s generally better to write out `maxSubstitutionRatePerSitePerYear` or `max.sub.rate.per.site.yr` than `maxSR`.
- However, very common variables should have short names.

- When helpful, incorporate identifiers like `df` (data frame), `ctr` (counter), or `idx` (index) into variable names to remind you of their purpose or structure.
- Customarily, variables that start with capital letters indicate global scope in `R`, and function names also start with capital letters. People argue about conventions, and they vary from language to language. Here's Google's style guide.

Do not use magic numbers. "Magic numbers" refer to numbers hard-coded in your functions and main program. Any number that could possibly vary should be in a separate section for parameters. The following code is not robust:

```
while ((age >= 5) && (inSchool == TRUE)) {
  yearsInSchool = yearsInSchool + 1
}
```

We may decide the age cutoff of 5 is inappropriate, but even if we never do, being unable to change the cutoff limits our ability to test the code. Better:

```
while ((age >= AgeStartSchool) && (inSchool == TRUE)) {
  yearsInSchool = yearsInSchool + 1
}
```

Most of the time, the only numbers that should be hard-coded are 0, 1, and pi.

Use labels for column and row names, and load functions by argument.

Don't force (or trust) yourself to remember that the first column of your data frame contains the time, the second column contains the counts, and so on. When reviewing code later, it's harder to interpret `cellCounts[,1]` than `cellCounts$time`.

In the same vein, if you have a function taking multiple inputs, it is safest to pass them in with named arguments. For instance, the function

```
BirthdayGiftSuggestion <- function(age, budget) {
  # ...
}
```

could be called with

```
BirthdayGiftSuggestion(age = 30, budget = 20)
# or
BirthdayGiftSuggestion(30, 20)
```

but the former is obviously safer.

Avoid repetitive code. If you ever find yourself copying and pasting a section of code, perhaps modifying it slightly each time, stop. It's almost certainly worth writing a function instead. The code will be easier to read, and if you find an error, it will be easier to debug.

Principle 3. Develop one function at a time, and test it before writing another.

The first part of this principle is easy for scientists to understand. When building code, we want to change one thing at a time. Controlled experiments are a great way to understand what's going on. Thus, we start by writing just a single function. It might not do exactly what we want it to do in the final program (e.g., it might contain mostly placeholders for functions it calls that we haven't written yet), but we want to be intimately familiar with how our code works in every stage of development.

The second part of this principle underscores one of the most important rules in defensive programming: **do not believe anything works until you have tested it thoroughly, and then keep your guard up.** *Expect* your code to contain bugs, and leave yourself time to play with the code (e.g., by trying to "break" it) until you can convince yourself they are gone. This involves an extra layer of defensive programming beyond the straightforward good practices discussed in Principle 2. Testing the code as you build it makes it much faster to find problems.

Unit tests. Unit tests are tests on small pieces of code, often functions. An intuitive and informal method of unit testing is to include `print()` statements in your code.

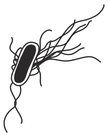
Here's a function to calculate the Simpson Index, a useful diversity index. It gives the probability that two randomly drawn individuals belong to the same species or type:

```r
SimpsonIndex <- function(C) {
  print(paste(c("Passed species counts:", C), collapse=" "))
  fractions <- C / sum(C)
  print(paste(c("Fractions:", round(fractions, 3)), collapse=" "))
  S <- sum(fractions ^ 2)
  print(paste("About to return S =",S))
  return(S)
}

# Simulate some data
numSpecies <- 10
maxCount <- 10 ^ 3
fakeCounts <- floor(runif(numSpecies, min = 1, max = maxCount))

# Call function with simulated data
S <- SimpsonIndex(fakeCounts)
```

```
## [1] "Passed species counts: 820 608 955 176 37 431 632 958 252 512"
## [1] "Fractions: 0.152 0.113 0.177 0.033 0.007 0.08 0.117 0.178 0.047 0.095"
## [1] "About to return S = 0.131756511386692"
```

It's very useful to print values to screen when you are writing a function for the first time and testing that one function. When you've drafted your function, I recommend walking through the function with `print()` and comparing the computed values to calculations you perform by hand or some other way. It can also be useful to do this at a very high level (more on that later).

The problem with relying on `print()` is that it rapidly provides too much information for you to process, and hence errors can slip through.



E. coli

Assertions

A more reliable way to catch errors is to use assertions. As someone from the internet once said: if you catch yourself thinking "this should and will always be true", use assertions to check for specific conditions in that case. Assertions are automated tests embedded in the code. The built-in function for assertions in `R` is `stopifnot()`. It's very simple to use.

Let's remove the `print` statements and add a check to our input data:

```
SimpsonIndex <- function(C) {
  stopifnot(C > 0)
  fractions <- C / sum(C)
  S <- sum(fractions ^ 2)
  return(S)
}
```

If each element of our abundances vector `C` is positive, `stopifnot()` will be `TRUE`, and the program will continue. If any element does not satisfy the criterion, then `FALSE` will be returned, and execution will terminate. Explore for yourself:

```
# Simulate two sets of data
numSpecies <- 10
maxCount <- 10 ^ 3
goodCounts <- floor(runif(numSpecies, min = 1, max = maxCount))
badCounts <- floor(runif(numSpecies, min = -maxCount, max = maxCount))

# Call function with each data set
S <- SimpsonIndex(goodCounts)
S <- SimpsonIndex(badCounts)
```

This gives a very literal error message, which is often enough when we are still developing the code. But what if the error might arise in the future, e.g., with future inputs? We can use the built-in function `stop()` to include a more informative message:

```
SimpsonIndex <- function(C) {
  if(any(C < 0)) stop("Species counts should be positive.")
  fractions <- C / sum(C)
  S <- sum(fractions ^ 2)
  return(S)
}
```

Now try it with `badCounts` again.

What about warnings? For instance, our calculation of the Simpson Index is an approximation: the index formally assumes we draw without replacement, but we have been computing $S = \sum p^2$, where p is the fraction of each species. It should be $S = \sum \frac{n(n-1)}{N(N-1)}$, where n is the abundance of each species n and N the total abundance. This simplification becomes important at small sample sizes. We could add a warning to alert users to this issue:

```r
SimpsonIndex <- function(C) {
  if(any(C < 0)) stop("Species counts should be positive.")
  if((mean(C) < 20) || (min(C) < 5)) {
    warning("Small sample size. Result will be biased. Consider corrected index.")
  }
  fractions <- C / sum(C)
  S <- sum(fractions ^ 2)
  return(S)
}

smallCounts <- runif(10)
S <- SimpsonIndex(smallCounts)
```

```
## Warning in SimpsonIndex(smallCounts): Small sample size. Result will be biased.
## Consider corrected index.
```

The main advantage of `warning()` over `print()` is that the message is red and will not be confused with expected results, and warnings can be controlled (see `?warning`).

You could make a case that `warning()` should be `stop()`. In general, with defensive programming, you want to halt execution quickly to identify bugs and to limit misuse of the code.

Exercise

- What other input checks would make sense with `SimpsonIndex()`?
- You can see how the code would be more readable and organized if most of that function were dedicated to actually calculating the Simpson Index. Draft a separate function, `CheckInputs()`, and include all tests you think are reasonable.
- When you and a neighbor are done, propose a bad or dubious input for their function and see if it's caught.

There are many packages that produce more useful assertions and error messages than what is built into `R`. See, e.g., `assertthat` and `testit`.

Exception handling

Warnings and errors are considered "exceptions." Sometimes it is useful to have an automated method to handle them. `R` has two main functions for this: `try()` allows you to continue executing a function after an error, and `tryCatch()` allows you to decide how to handle the exception.

Here's an example:

```
UsefulFunction <- function(x) {
  value <- exp(x)
  otherStuff <- rnorm(1)
  return(list(value, otherStuff))
}

data <- "2"
results <- UsefulFunction(data)
print(results)
```

Now `results` is quite a disappointment: it could've at least returned a random number for you, right? You could instead try

```
UsefulFunction <- function(x){
  value <- NA
  try(value <- exp(x))
  otherStuff <- rnorm(1)
  return(list(value, otherStuff))
}
results <- UsefulFunction(data)
```

```
## Error in exp(x) : non-numeric argument to mathematical function
```

```
print(results)
```

```
## [[1]]
## [1] NA
##
## [[2]]
## [1] 0.830212
```

Even though the function still can't exponentiate a string (`exp("2")` still fails), execution doesn't terminate. If we want to suppress the error message, we can use `try(..., silent=TRUE)`. This obviously carries some risk!

We could make this function even more useful by handling the error responsibly with `tryCatch()`:

```
UsefulFunction <- function(x){
  value <- NA
  tryCatch ({
      message("First attempt at exp()...")
      value <- exp(x)},
    error = function(err){
      message(paste("Darn:", err, " Will convert to numeric."))
      value <<- exp(as.numeric(x))
      }
    )
  otherStuff <- rnorm(1)
  return(list(value, otherStuff))
}
results <- UsefulFunction(data)
print(results)
```

It is also possible to assign additional blocks for warnings (not just errors). The `<<-` is a way to assign to the value in the environment one level up (outside the `error=` block).

> **Exercise**
>
> The package ggridges works with package `ggplot2` to show multiple distributions in a superimposed but interpretable way. Let's say we want to run the following code:

```
library(ggplot2)
library(ggridges)
ggplot(diamonds, aes(x = price, y = cut, fill = cut, height = ..density..)) +
  geom_density_ridges(scale = 4, stat = "density") +
  scale_y_discrete(expand = c(0.01, 0)) +
  scale_x_continuous(expand = c(0.01, 0)) +
  scale_fill_brewer(palette = 4) +
  theme_ridges() + theme(legend.position = "none")
```

> You probably don't have `ggridges` installed yet, so you'll get an error. Use `tryCatch()` so that the package is installed if you do not have it and then loaded.

Test all the scales!

It's important to consider multiple scales on which to test as you develop. We've focused on unit tests (testing small functions and steps) and testing inputs, but it is easy to have correct subroutines and incorrect results. For instance, we can be excellent at the distinct skills of toasting bread, buttering bread, and eating bread, but we will fail to enjoy buttered toast for breakfast if we don't pay attention to the order.

With scientific programming, it is critical to simplify code to the point where results can be compared to analytic expectations. You saw this in Principle 1. It is important to add functions to check not only inputs and intermediate results but also larger results. For instance, when we set the birth rate equal to the death rate, does the code reliably produce a stable population? We can write functions to test for precisely such requirements. These are **system tests**. When you change something in your code, always rerun your system tests to make sure you've not messed something up. Often it's helpful to save multiple parameter sets or data files precisely for these tests.

It's hard to overstate the importance of taking a step back from the nitty-gritty of programming and asking, Are these results reasonable? Does the output make sense with different sets of extreme values? Schedule time to do this, and update your system tests when necessary. Please don't expect your collaborators to do this work for you (unless you've arranged to trade this kind of help).

Principle 4. Document often.

It is helpful to keep a running list of known "issues" with your code, which would include the functions left to implement, the tests left to run, any strange bugs/behavior, and features that might be nice to add later. Sites like GitHub and Bitbucket allow you to associate issues with your repositories and are thus very helpful for collaborative projects, but use whatever works for you. Having a formal to-do list, however, is much safer than sprinkling to-do comments in your code (e.g., `# CHECK THIS!!!`). It's easy to miss comments.

Research code will always need a `readme` describing the software's purpose and implementation. It's easiest to develop it early and update as you go.

In R, the standard for documenting the code is called `roxygen2`. Upon pressing Ctrl+Shift+option+R, RStudio automatically generate a little roxygen2 skeleton for a local piece of function. We can fill in different fields with basic information about our function.

Principle 5. Refactor often.

To refactor code is to revise it to make it clearer, safer, or more efficient. Because it involves no changes in the scientific outputs (results) of the program, it might feel pointless, but it's usually not. Refactor when you realize that your variable and function names no longer reflect their true content or purpose (rename things quickly with `Ctrl + Alt + Shift + M`), when certain functions are obsolete or should be split into two, when another data structure would work dramatically better, etc. Any code that you'll be working with for more than a week, or that others might ever use, should probably be refactored a few times. Debugging will be easier, and the code will smell better.

Important tip, repeated: Run unit and system tests after refactoring to make sure you haven't messed anything up. This happens more than you might think. You can even automate/enforce tests to run in GitHub!

Principle 6. When you run your code, save each "experiment" in a separate directory with a complete copy of the code repository and all parameters.

When you're done developing the code and are using it for research, keep results organized by creating a separate directory for each execution of the code that includes not only the results but also the precise code used to generate the results. This way, you won't accidentally associate one set of parameters with results that were in fact generated by another set of parameters. Here's a sample workflow, assuming your repository is located remotely on GitHub, and you're in a UNIX terminal:

```
$ mkdir 2021-09-13_rho=0.5
$ cd 2021-09-13_rho=0.5
$ git clone git@github.com:MyName/my-repo
```

If we want, we can edit and execute our code from within `R` or `RStudio`, but we can also keep going with the command line. Here we are using a built-in UNIX text editor known as emacs. If you are a glutton for punishment, you could instead use vi(m). (Current Mac OS users will need to use vim or nano unless they install emacs separately.)

```
$ cd my-repo
$ emacs parameters.json // (edit parameters, with rho=0.5)
$ Rscript mycode.R
```

Keeping your experiments separate is going to save your sanity for larger projects when you repeatedly revise your analyses. It also makes isolating bugs easier.

At this point, we can attempt to argue that when different versions or experiments of your code won themselves a separate directory (this will be the case all the time as you constantly modify your analysis throughout your projects), save those different versions in a package-style. Pick the best documented R package you know and go to its source code package, and you will find that it typically contains the followings:

1. A DESCRIPTION and NAMESPACE file, both have equivalent role as the above-mentioned `readme` file.
2. Of course, the source code, including comments and documentation in `roxygen2` style.
3. Manage dependencies, preferably in a "packrat" repo. A packrat project has its own private package library. Any packages you install from inside a packrat project are only available to that project; and packages you install outside of the project are not available to the project. *Note: renv is a stable replacement for packrat.*
4. The documentation, helping users to understand the code and in particular, if the code is to be part of a pipeline, explaining how to interact with the API it exposes. Where the work product is an analysis rather than a bit of code intended to carry out a task, writing a vignettes.

For more details, see this article by Chris.

Principle 7. Don't be too *defensive.*

This is not necessary:

```
myNumbers <- seq(from = 1, to = 500, length.out = 20)
stopifnot(length(myNumbers) == 20)
```

It's fine to check stuff like this when you're getting the hang of a function, but it doesn't need to be in the code. Code with too many defensive checks becomes a pain to read (and thus debug). Try to find a balance between excess caution and naive optimism. Good luck!

Part 2: Debugging in R

Part 1 introduced principles that should minimize the need for aggressive debugging. You're in fact already debugging if you're regularly using input, unit, and system tests to make sure things are running properly. But what happens when despite your best efforts, you're not getting the right result?

We'll focus on more advanced debugging tools in this part. First, some general guidelines for fixing a bug:

- *Isolate the error and make it reproducible.* Try to strip the error down to its essential parts so you can reliably reproduce the bug. If a function doesn't work, copy the code, and keep removing pieces that are non-essential for reproducing the error. When you post for help on the website stackoverflow, for instance, people will ask for a MRE or MWE—a minimum reproducible (working) example. You need to have not only the pared code but also the inputs (parameter values and the seeds of any random number generators) that cause the problem.
- *Use assertions and debugging tools to hone in on the problem.* We've already seen how to use `stop()` and `stopifnot()` to identify logic errors in unit tests. We'll cover more advanced debugging here.
- *Change/Test one thing at a time.* This is why we develop only one function at a time.

Note: Learning how to make MREs and how to find the relevant parts of code is especially applicable when finding bugs/issues in other people's code (even if it isn't R). The better your issue description is, the more likely the author's will be willing to help fix it! You can see an example of a bug I found in the Python package `pysam` and the issue I posted on GitHub here.

Tracing calls

If an error appears, a useful technique is to see the call stack, the sequence of functions immediately preceding the error. We can do this in R using `trackback()` or in RStudio.

The following example comes from a nice tutorial by Hadley Wickham:

```
f <- function(a) g(a)
g <- function(b) h(b)
h <- function(c) i(c)
i <- function(d) "a" + d
f(10)
```

You can see right away that running this code will create an error. Try it anyway in RStudio. Click on the "Show Traceback" to the right of the error message. What you're seeing is the stack, and it's helpfully numbered. At the bottom we have the most recent (proximate) call that produced the error, the preceding call above it, and so on. (If you're working from a separate R file that you sourced for this project, you'll also see the corresponding line numbers next to each item in the stack.) If you're in R, you can run `traceback()` immediately after the error.

Seeing the stack is useful for checking that the correct functions were indeed called, but it can suggest how to trace the error back in a logical sequence. But we can often debug faster with more information from RStudio's debugger.

Examining the environment

Let's pretend we have a group of people who need to be assigned random partners. These partnerships are directed (so person A may consider his partner person B, but B's partner is person F), and we'll allow the possibility of people partnering with themselves. Some code for this is in the file `BugFun.R`. (It's not terribly efficient code, but it is useful for this exercise.)

```r
source("BugFun.R")
peopleIDs <- seq(1:10)
pairings <- AssignRandomPartners(peopleIDs)
```

Try running this a few times. We have an inconsistent bug.

We can use breakpoints to quickly examine what's happening at different points in the function. With breakpoints, execution stops on that line, and environmental states can be inspected. In RStudio, you can create breakpoints by clicking to the left of the line number in the `.R` file. A red dot appears. You can then examine the contents of different variables in debug mode. To do this, you have to make sure you have the right setting defined: Debug > On Error > Break in Code.

> **Exercise**
>
> Use breakpoints to identify the error(s) in the AssignRandomPartners() function. Go to `BugFun.R` and attempt to run the last line. Decide on a place to start examining the code. If you have adjusted your settings, the debug mode should start automatically once you define a breakpoint and try to run the code again. (If a message to source the file appears, follow it.) Your console should now have `Browse[1]>` where it previously had only >.
>
> The IDE is now giving you lots of information. The green arrow shows you where you are in the code. The line that is about to be executed is in yellow. Anything you execute in the console shows states from your current environment. Test this for yourself by typing a few variables in the console. You can see these values in the Environment pane in the upper right, and you can also see the stack in the middle right.
>
> If you hit enter at the console, it will advance you to the next step of the code. But it is good to explore the Console buttons (especially 'Next') to work through the code and watch the Environment (data and values) and Traceback as they change. By calling the function several times, you should be able to convince yourself of the cause of the error.
>
> Much more detail about browsing in debugging mode is available here.
>
> When you are done, exit the debug mode by hitting the Stop button in the Console.

In R, you can insert the function `browser()` on some line of the code to enter debugging mode. This is also what you have to use if you want to debug directly in R Markdown.

Examining the source code

The most effective way to debug when using functions from R package is to download their source code, and check the code directly to see for yourself if there is anything that goes wrong.

My 2 cents of advice from my self-taught programming experience: don't reinvent the wheel when your primary goal is to do good research, simply borrow the wheel from experts who focus on pure software development. I learned programming in different languages by reading the source code. Reading source code allows me to assimilate others' good style of writing their code, and allows me to build a very clear connection between errors that I encounter with the parts in code responsible for those errors. Throughout the process, I also get a sense of how my system and a specific programming language works, as much as I could. I am also able to learn the failure modes of the language when I break its certain component, for example, the association between what I have just changed recently, and what the downstream impact of my change in the cascade of my entire code network could be. Eventually I am able to narrow down the chance of producing any error in my code as I improve my programming skill, and even when an error happens, I will have a good hunch about what could go wrong immediately.

Programming Challenge

Avian influenza cases in humans usually arise from two viral subtypes, H5N1 and H7N9. An interesting observation is that the age distributions for H5N1 and H7N9 cases differ: older people are more likely to get very sick and die from H7N9, and younger people from H5N1. There's no evidence for age-related differences in exposure. A recent paper showed that the risk of severe infection or death with avian influenza from 1997-2015 could be well explained by a simple model that correlated infection risk with the subtype of seasonal (non-avian) influenza a person was first exposed to in childhood. Different subtypes (H1N1, H2N2, and H3N2) have circulated in different years. Perhaps because H3N2 is more closely related to H7N9 than to H5N1, people with primary H3N2 infections seem protected from severe infections with H7N9. The complement is true for people first infected with H1N1 or H2N2 and later exposed to H5N1.

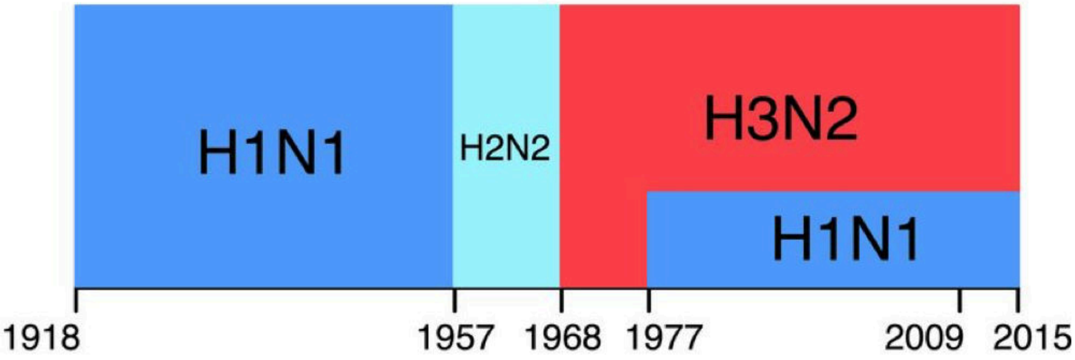


Figure 1: *Endemic influenza subtypes since 1918 (Gostic et al. 2016).*

Of course, we do not know the full infection history of any person who was hospitalized with avian influenza. We only know the person's age and the year of hospitalization or death. To perform their analysis, the authors needed to calculate the probability that each case had a primary infection with each subtype, i.e., the probability that a person born in a given year was first infected with each subtype. Your challenge is to calculate these probabilities.

The authors had to make some assumptions. First, they assumed that the risk of influenza infection is 28% in each year of life. Second, they assumed that the frequency of each circulating subtype could be inferred from the numbers of isolates sampled (primarily in hospitals) each year. These counts are given in `subtype_counts.csv`. [1]

The challenge: For every year between 1960 and 1996, calculate the probability that a person born in that year had primary influenza infection with H1N1, H2N2, and H3N2. You must program defensively to pull this off.

[1] The counts are actually given for each influenza season in the U.S., which is slightly different

from a calendar year, but you can ignore this. You'll notice that "1" and "0" are used where we know (or assume) that only one subtype was circulating. The authors made several other assumptions, but this is good enough for now.

Statistics for a data rich world—some explorations[*]

Stefano Allesina (adapted by Lin Chen) *University of Chicago*

Goal: More and more often we need to analyze large and complex data sets. However, the statistical methods we've been taught in college have evolved in a data-poor world. Modern biology requires new tools, which can cope with the new questions and methods that arise in a data-rich world. Here we are going to discuss problems that often arise in the analysis of large data sets. We're going to review hypothesis testing (and what happens when we have many hypotheses) and discuss model selection. We're going to see the effects of selective reporting and p-hacking, and how they contribute to the *reproducibility crisis* in the sciences. **Audience:** Biologists with some programming background.

I. Review of hypothesis testing

Statistics is the science of collecting and analyzing data from samples in order to estimate and make inference regarding the the population. A census data is often not feasible, and also not necessary. A sample is a smaller and random subset of the target population and the sample is collected to represent the population. Hypothesis testing is a major component of statistical inference.

The basic idea of hypothesis testing is the following: we have devised an hypothesis on our system, which we call H_1 (the "alternative hypothesis"). We have collected our data, and we would like to test whether the data are consistent (or inconsistent) with the so-called "null hypothesis" (H_0). The null hypothesis is a contradiction to the alternative hypothesis.

The simplest example is that of a bent coin: my friend Aiden likes to bet on a coin toss with people, and he often chooses head and wins. I suspect that he has a bent coin in favor of head (H_1: the coin is bent). We therefore toss the coin several times and check whether the number of heads we observe is consistent with the null hypothesis of a fair coin (H_0: the coin is a fair coin).

In `R` we can toss many coins in no time at all. Call `p` the probability of obtaining a head, and initially toss a fair coin (`p = 0.5`) a thousand times:

First, when simulating a data, we always want to set seed to make sure the results are reproducible.

```r
set.seed(222)
p <- 0.5 # probability of a head (fair coin)
flips <- 1000 # number of times we flip the coin
data <- sample(c("H", "T"),
               flips, prob = c(p, 1 - p),
               replace = TRUE)
heads <- sum(data == "H")
```

[*] This document is included as part of the 'Statistics for a data rich world' tutorial packet for the BSD qBio Bootcamp, University of Chicago, 2021. **Current version**: August 26, 2021. **Corresponding author**: sallesina@uchicago.edu. Adapted by Lin Chen (lchen@health.bsd.uchicago.edu).

There is also a faster way to flip the coins and count the heads by assuming that the number of heads out of 1000 flips follows a binomial distribution:

```r
heads <- rbinom(1, flips, p)
```

If the coin is fair, we expect approximately 500 heads, but of course we might have small variations due to the randomness of the process. We therefore need a way to distinguish between "bad luck" and an incorrect hypoythesis.

What is customarily done is to compute the probability of recovering the observed or a more extreme version of the pattern under the null hypothesis: if the probability is very small, it implies that when the null hypothesis is true, there is a very small probability (i.e., very unlikely) that you can observe things as or more extreme than what you have observed in the current data. We call this probability a *p-value*. A small *p-value* (typically less than 0.05) indicates strong evidence against the null hypothesis, so you reject the null hypothesis and conclude that the data is inconsistent with the null hypothesis. A large *p-value* (> 0.05) only means that when the null hypothesis is true, you are likely to observe what has been observed in the current data but that is not enough to prove the null hypothesis is true, so you fail to reject the null hypothesis and the conclusion is inconclusive. When the null hypothesis is indeed true or when the alternative hypothesis is true but you do not have enough power to reject the null, you may end up with a large p-value.

For example, if the coin is fair, the number of heads should follow the binomial distribution. The probability of observing a larger number of heads than what we've got is therefore

```r
one.sided.pvalue <- 1 - pbinom(heads, flips, 0.5)
```

Here and in the following sections, we calculated a one-sided p-value testing $H_0 : p = 0.5$ versus $H_A : p > 0.5$. Note that in many other cases, we recommend a two-sided test, and a two-sided p-value together with a 95% confidence interval can be obtained via:

```r
heads <- rbinom(1, flips, p)
pvalue <- binom.test(heads, flips, 0.5, alternative="two.sided")
```

What if we repeat the tossing many, many times?

```r
# flip 1000 coins 1000 times, and count the number of heads in each of the 1000 experiments
# produce histogram of number of heads
heads_distribution <- rbinom(1000, flips, p)
hist(heads_distribution, main = "distribution number heads", xlab = "number of heads")
```

You can see that it is very unlikely to get more than 560 (or less than 440) heads when flipping a fair coin 1000 times. Therefore, if we were to observe say 400 heads (or 600), we would tend to believe that the coin is biased (though of course this could have happened by chance if we are repeating the tossing 1 trillion times!).

Type I and type II errors

When testing an hypothesis, we can make two types of errors:

- **Type I error**: reject H_0 when it is in fact true. Also known as false positive.
- **Type II error**: fail to reject H_0 when in fact it is not true. Also known as false negative.

We call α the probability of making a type I error (or type I error rate), and β as type II error rate. And power is in fact $1 - \beta$. We can calculate the p-value based on the data and compare the p-value with the significance threshold α. The p-value quantifies how strongly the data contradicts the null hypothesis, and if $p < \alpha$, we reject the null hypothesis. Type I and Type II error rates are inversely related. In choosing a significance level for a test, you are actually deciding how much you want to risk committing a type I error — rejecting the null hypothesis when it is. The more stringent α is (0.01 versus 0.05) in controlling type I error rate, the less likely you would make a rejection decision and consequently the power will be reduced too.

The distribution of p-values

Suppose that we are tossing each of several fair coins 1000 times. For each, we compute the corresponding (one-sided) p-value testing the null hypothesis `p = 0.5` against the alternative `p > 0.5`. How are the p-values distributed?

```r
ncoins <- 2500
heads <- rbinom(ncoins, flips, p)
pvalues <- 1-pbinom(heads, flips, 0.5)
hist(pvalues, xlab = "p-value", freq = FALSE)
abline(h = 1, col = "red", lty = 2)
```

As you can see, if the data were generated under the null hypothesis, the distribution of the p-values would be approximately uniform between 0 and 1. This means that if we set $\alpha = 0.05$, we would reject the null hypothesis 5% of the time (even though in this case we know the hypothesis is correct!).

What is the distribution of the p-values if we are tossing biased coins? We will find an enrichment in small p-values, with stronger effects for larger biases:

```r
p <- 0.52 # the coin is biased
heads <- rbinom(ncoins, flips, p)
pvalues <- 1 - pbinom(heads, flips, 0.5)
hist(pvalues, xlab = "p-value", main = paste0("p =  ", p), freq = FALSE)
abline(h = 1, col = "red", lty = 2)

p <- 0.55 # the coin is biased
heads <- rbinom(ncoins, flips, p)
pvalues <- 1 - pbinom(heads, flips, 0.5)
hist(pvalues, xlab = "p-value", main = paste0("p =  ", p), freq = FALSE)
abline(h = 1, col = "red", lty = 2)
```

II. The challenges with p-values

Selective reporting

Articles reporting positive results are easier to publish than those containing negative results. Authors might have little incentive to publish negative results, which could go directly into the file-drawer.

This tendency is evidenced in the distribution of p-values in the literature: in many disciplines, one finds a sharp decrease in the number of tests with p-values just below 0.05 (which is customarily–and arbitrarily–chosen as a threshold for "significant results"). For example, we find many a sharp decrease in the number of reported p-values of 0.051 compared to 0.049–while we expect the p-value distribution to decrease smoothly.

Selective reporting leads to irreproducible results: we always have a (small) probability of finding a "positive" result by chance alone. For example, suppose we toss a fair coin many times, until we find a "signficant" result.

On the other hand, more and more journals would require us to report effect size estimates and confidence intervals – "Were this procedure to be repeated on numerous samples, the fraction of calculated confidence intervals (which would differ for each sample) that encompass the true population parameter would tend toward 90%."(source: Cox D.R., Hinkley D.V. (1974) Theoretical Statistics, Chapman & Hall, p49, p209.)

Problem: p-hacking

The problem is well-described by Simonsohn et al. (J. Experimental Psychology, 2014): "While collecting and analyzing data, researchers have many decisions to make, including whether to collect more data, which method to use, which measure(s) to analyze, which covariates to use, what to do with outliers and missing data, and so on. If these decisions are not made in advance but rather are made as the data are being analyzed, then researchers may make them in ways that self-servingly increase their odds of publishing. Thus, rather than placing entire studies in the file-drawer, researchers may file merely the subsets of analyses that produce nonsignificant results. We refer to such behavior as *p-hacking*." The term p-hacking describes the conscious or subconscious manipulation of data in a way that produces a desired p-value.

The same authors showed that with careful p-hacking, almost anything can become significant (read their hylarious article in Psychological Science, where they show that listening to a song can change the listeners' age!).

Discussion on p-values

Selective reporting and p-hacking are only two of the problems associated with the widespread use and misuse of p-values. The discussion in the scientific community on this issue is extremely topical. I have collected some of the articles on this problem in the `readings` folder. Importantly, in 2016 the American Statistical Association released a statement on p-values every scientist should read.

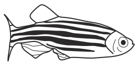
Reproducibility crisis

P-values and hypothesis testing contribute considerably to the so-called *reproducibility crisis* in the sciences. A survey promoted by *Nature* magazine found that "More than 70% of researchers have tried and failed to reproduce another scientist's experiments, and more than half have failed to reproduce their own experiments."

This problem is due to a number of factors, and addressing it will likely be one of the main goals of science in the next decade.

Exercise: p-hacking

Go to goo.gl/a3UOEF and try your hand at p-hacking, showing that your favorite party is good (bad) for the economy.



D. rerio

III. Multiple comparisons (also known as multiple testing)

The problem of multiple comparisons arises when we perform multiple statistical tests. Since each test is subject to some small chance of producing a false positive result, when jointly considering many many tests, the chances of producing some false positive findings are much higher.

Suppose we perform our coin tossing exercise, flipping 1000 coins 1000 times each. For each coin, we determine whether our data differs significantly from what expected by contrasting our p-value with a significance level $\alpha = 0.05$.

Even if the coins are all perfectly fair, we would expect to find approximately $0.05 \cdot 1000 = 50$ coins that lead to the rejection of the null hypothesis.

In fact, we can calculate the probability of making at least one type I error (reject the null when in fact it is true). This probability is called the Family-Wise Error Rate (FWER). It can be computed as 1 minus the probability of making no type I error at all. If we set $\alpha = 0.05$, and assume the tests to be independent, the probability of making no errors in m tests is $1 - (1 - 0.05)^m$. Therefore, if we perform 10 tests, we have about 40% probability of making at least a mistake; if we perform 100 tests, the probability grows to more than 99%. If the tests are not independent, we can still say that in general $FWER \leq m\alpha$.

This means that setting an α per test does not control for FWER.

Moving from tossing coins to biology, consider the following examples:

- **Gene expression** In a typical RNAseq experiment, we compare the differential expression levels of tens of thousands of genes in the treatment and control tissues.
- **GWAS** In Genome-Wide Association Studies we want to find single-nucleotide polymorphisms (SNPs) associated with a given phenotype. It is common to test millions of SNPs for signficant associations.
- **Identifying binding sites** Identifying candidate binding sites for a transcriptional regulator requires scanning the whole genome, yielding tens of millions of tests.

Organizing the tests in a table

Suppose that we're testing m hypotheses. Of these, an unknown subset m_0 is true, while the remaining $m_1 = m - m_0$ are false. We would like to correctly call the true/false hypotheses (as much as possible). We can summarize the results of our tests in a table, of which the elements are unobservable:

What we would like to know is $m_1 = T + S$ and $m_0 = U + V$. Then V is the number of type I errors (rejected H when in fact it is true), and T is the number of type II errors (failed to reject a false H). However, we can only observe $V + S$ (the number of "discoveries"), and $U + T$ (number of "failures").

The type I error rate is $E[V]/m$ (where $E[X]$ stands for expectation). When there are many tests (m) being considered, controlling for type I error rates at 0.05 means that by random chance, one could make $0.05 \times m$ false positive findings even if all m tests are under the null. For example, when testing genetic associations between 10 million genetic variants to the risk of a disease, if still using 0.05 as the significance threshold, there could be 500k significant findings by random chance even if the disease is not heritable and has no genetic association. Apparently, we need to choose a much more strigent significance threshold to account for the number of tests, and there are some other error measures. The Family-wise error rate is defined as $P(V > 0)$. Another quantity of interest is the False Discovery Rate (FDR), measured as the proportion of true discoveries $FDR = E[V/(V + S)]$ when $V + S > 0$. FDR measures the proportion of falsely rejected hypotheses.

Importantly, FWER guards against any single false positive finding among all m tests, and is a more stringent significance criteria than FDR in multiple comparison problems. It also means that when we control for FWER, we're automatically controlling for the FDR, but not vice versa. It should be noted that when a large number of tests is performed, controlling FWER could be quite conservative and may lose power.

Methods for multiple testing correction: Bonferroni correction

One of the simplest and most widely-used procedures to control for FWER is Bonferroni's correction. This procedure controls for FWER in the case of independent or dependent tests. It is typically quite conservative, especially when the tests are not independent (in practice, it becomes "too conservative" when the number of tests is moderate to high). Fundamentally, for a desired FWER α we choose as a significance treshold of α/m for each single test, where m is the number of tests we're performing. Equivalently, we can "adjust" the p-values as $q_i = \min(m \cdot p_i, 1)$, and call significant the values $q_i < \alpha$. In `R` it is easy to perform this correction:

```r
original_pvals <- c(0.012, 0.06,  0.77, 0.001, 0.32)
adjusted_pvals <- p.adjust(original_pvals, method = "bonferroni")
print(adjusted_pvals)
```

With these adjusted p-values, and an $\alpha = 0.05$, we would still single out as significant the fourth test, but not the first. The strength of Bonferroni is its simplicity, and the fact that we can perform the operation in a single step. Moreover, the order of the tests does not matter.

Other procedures: the Holm–Bonferroni method (or the Holm method)

There are several refinements of Bonferroni's correction, some of which use the sequence of ordered p-values. For example, the Holm method starts by sorting the p-values in increasing order $p_{(1)} \leq p_{(2)} \leq p_{(3)} \leq \ldots p_{(m)}$. The hypothesis $H_{(i)}$ is rejected if $p_{(j)} \leq \alpha/(m-j+1)$ for all $j = 1, \ldots, i$. Equivalently, we can adjust the p-values as $q_{(i)} = \min(1, \max((m-i+1)p_{(i)}, q_{(i-1)}))$. In this way, we use the most stringent threshold to determine whether the smallest p-value is significant, the next smallest p-value uses a slightly higher threshold and so on. The Holm method is uniformly more powerful than the Bonferroni correction.

For example, using the same p-values above:

```
original_pvals <- c(0.012, 0.06,  0.77, 0.001, 0.32)
adjusted_pvals <- p.adjust(original_pvals, method = "holm")
print(adjusted_pvals)
```

We see that we would be calling the first test significant, contrary to what obtained with Bonferroni.

The function `p.adjust` offers several choices for p-value correction. Also, the package `multcomp` provides a quite comprehensive set of functions for multiple hypothesis testing.

An example: testing mixed coins

We're going to test these concepts by tossing repeatedly many coins. In particular, we're going to toss 1000 times 50 biased coins ($p = 0.55$) and 950 fair coins ($p = 0.5$). For each coin, we're going to compute a p-value, and count the number of type I, type II, etc. errors when using unadjusted p-values as well as when correcting using the Bonferroni or Holm procedure.

```
toss_coins <- function(p, flips){
  # toss a coin with probability p of landing on head several times
  # return a data frame with p, number of heads, pval and
  # H0 = TRUE if p = 0.5 and FALSE otherwise
  heads <- rbinom(1, flips, p)
  pvalue <- 1 - pbinom(heads, flips, 0.5)
  if (p == 0.5){
    return(data.frame(p = p, heads = heads, pval = pvalue, H0 = TRUE))
  } else {
    return(data.frame(p = p, heads = heads, pval = pvalue, H0 = FALSE))
  }
}

# To ensure everybody gets the same results, we're setting the seed
set.seed(8)
data <- data.frame()
# the biased coins
for (i in 1:50) data <- rbind(data, toss_coins(0.55, 1000))
# the fair coins
```

```r
for (i in 1:950) data <- rbind(data, toss_coins(0.5, 1000))
# here's the data structure
head(data)
```

Now we write a function that adjusts the p-values and builds the table above

```r
get_table <- function(data, adjust, alpha = 0.05){
  # produce a table counting U, V, T and S
  # after adjusting p-values for multiple comparisons
  data$pval.adj <- p.adjust(data$pval, method = adjust)
  data$reject <- FALSE
  data$reject[data$pval.adj < alpha] <- TRUE
  return(table(data[,c("reject","H0")]))
}
```

First, let's see what happens if we don't adjust the p-values:

```r
no_adjustment <- get_table(data, adjust = "none", 0.05)
print(no_adjustment)
```

We correctly declared 48 of the biased coins "significant", but we also incorrectly called 2 biased coins "not significant" (Type II error). More worringly, we called 45 fair coins biased when they were not (Type I error). To control for the family-wise error rate, we can correct using Bonferroni:

```r
bonferroni <- get_table(data, adjust = "bonferroni", 0.05)
print(bonferroni)
```

With this correction, we dramatically reduced the number of type I errors (from 45 to 0), but at the cost of increasing type II errors (from 2 to 40) and losing power. In this way, we would make only 10 discoveries instead of 50.

In this case, Holm's procedure does not help:

```r
holm <- get_table(data, adjust = "holm", 0.05)
print(holm)
```

More sophisticated methods, for example the Benjamini-Hochberg (BH) procedure based on controlling false discovery rate (FDR=E(false discoveries/signifiant tests), where E stands for expectation), can reduce the type II errors and improve power, at the cost of a few and estimable type I errors:

```r
BH <- get_table(data, adjust = "BH", 0.05)
print(BH)
```

C. jacchus

FDR and q-values

Inspired by the need for controlling for FDR in genomics, Storey and Tibshirani (PNAS 2003) have proposed the idea of a q-value, measuring the probability that a feature that we deemed significant turns out to be not significant after all.

One uses p-values to control for the false positive rate (# false positive / total test): when determining significant p-values we control for the rate at which null features in the data are called significant. The False Discovery Rate (# false positive / # signficant test), on the other hand, measures the rate at which results that are deemed significant are truly null. While setting $PCER = 0.05$ we are stating that about 5% of the truly null features will be called significant, an $FDR = 0.05$ means that among the features that are called significant, about 5% will turn out to be null.

They proposed a method that uses the ensamble of p-values to determine the approximate (local) FDR. The idea is simple. If you plot your histogram of p-values when you have few true effect, and many nulls, you will see something like:

```r
hist(data$pval, breaks = 25)
```

where the right side of the histogram is close to a uniform distribution. We could use the high p-values to find how tall the histogram would be if all effects were null, thereby estimating the proportion of truly null features $\pi_0 = m_0/m$.

Storey has built an `R`-package for this type of analysis:

```r
# To install:
#install.packages("devtools")
#library("devtools")
#install_github("jdstorey/qvalue")
library("qvalue")
qobj <- qvalue(p = data$pval)
```

Here's the estimation of the π_0

```r
hist(qobj)
```

which is quite good (in this case we know that $\pi_0 = 0.95$). The small p-values under the dashed line represent our false discoveries. Even better, through randomizations one can associate a q-value to each test, representing the probability of making a mistake when calling a result significant (formally, the q-value is the minimum FDR that can be attained when calling that test significant).

For example:

```r
table((qobj$pvalues < 0.05) & (qobj$qvalues < 0.05), data$H0)
```

Note that the estimation of FDR is unstable if the demoniator (# significant test) is expected to be small. Therefore, you may notice that FDR was widely used in detecting differentially expressed genes in diseased versus normal samples where the expected number of non-null tests is large. In contast, in GWAS, researchers use the Bonferroni-adjusted p-value threshold of 5×10^{-8} to declare significance.

IV. Linear regression, logistic regression, and model selection

Linear regression

In statistics, linear regression is an approach to model the linear relationship between one response variable (or dependent variable) and one or more explanatory variables (or independent variables, or predictor). If there is only one exlanatory variable, it is called simple linear regression. If the linear regression involves more than one explanatory variable, it is a multiple linear regression.

```r
# create fake data for a simple linear regression
set.seed(5)
x <- 1:20
y <- 3 + 0.5 * x + rnorm(20)
plot(y ~ x)
```

We can fit a simple linear regression to the data

```r
model1 <- lm(y ~ x)
summary(model1)
plot(y~x)
points(model1$fitted.values~x, type = "l", col = "blue")
```

In a data rich world, often, we need to select a model out of a set of reasonable alternatives with different combinations and/or (even nonlinear) patterns of explanatory variables. However, we run the risk of overfitting the data (i.e., fitting the noise as well as the pattern). The best fitted model for the current data from the current sample may not be the best model representing the pattern in the population of interest. Here is a simple example of a overfitted regression:

For the above data, we can also fit a more complex polynomial function of x.

```r
model2 <- lm(y ~ poly(x, 7))
```

Let's see the residuals etc.

```r
summary(model1)
summary(model2)
plot(y~x)
points(model1$fitted.values~x, type = "l", col = "blue")
points(model2$fitted.values~x, type = "l", col = "red")
```

Our second model has a much greater R^2, i.e., more variation in the response variable can be explained by the model, but the second model also has many more parameters. The first model is more parsimonious. Which is a model we should choose?

Model selection tries to address this and similar problems. Most model fitting and model selection procedures are based on likelihoods (e.g., Bayesian models, maximum likelihood, minimum description length). The likelihood $L(\theta|D, M)$ is (proportional to) the probability of observing the data D under the model M and parameters θ. Because likelihood can be very small when you have much data, typically one works with log-likelihoods. For example:

```
logLik(model1)
logLik(model2)
```

Typically, more complex models will yield better (less negative) log-likelihoods and a better fit for the current data. However, more parameters would also increase the variation of the model. A complex model may not best represent the population (not to say computational burden). We will need to find a balance between bias and variance. We therefore want to penalize more complex models in some way.

AIC

One of the simplest methods to select among competing models is the Akaike Information Criterion (AIC). It penalizes models accoding to the **number of parameters**: $AIC = 2p - 2\log L(\theta|D,M)$, where p is the number of parameters. Note that **smaller** values of AIC stand for "better" models. In R you can compute AIC using:

```
AIC(model1)
AIC(model2)
```

As you can see, AIC would favor the first (and simpler) model, which is also the model we used to simulate the data.

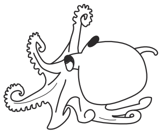
AIC is rooted in information theory and measures (asymptotically) the loss of information when using the model instead of the data. There are several limitations of AIC: a) it only holds asymptotically (i.e., for very large data sets; for smaller data you need to correct it); it penalizes each parameter equally (i.e., parameters that have a large influence on the likelihood have the same weight as parameters that do no influence the likelihood much); it can lead to overfitting, favoring more complex models in simulated data generated by simpler models.

BIC

In a similar vein, BIC (Bayesian Information Criterion) uses a slightly different penalization: $BIC = \log(n)p - 2\log L(\theta|D,M)$, where n is the number of data points. You may see that for large data, BIC penalizes a complex model more than AIC. Again, smaller values stand for "better" models:

```
BIC(model1)
BIC(model2)
```

Here in this simple example, AIC and BIC agree with each other, and the simpler model is favored.



O. bimaculoides

Logistic regression

Logistic regression is often used to model the nonlinear relationship (modeled by a logistic function) between a binary response variable and a linear combination of explantory variables. When the outcome is binary, for example pass/fail, win/lose, alive/dead, yes/no, one would consider a logistic regression. It can be extended to model several classes of a categorical variable as response.

We will start with an example. Fox et al. (Research Integrity and Peer Review, 2017) analyzed the invitations to review for several scientific journals, and found that "The proportion of invitations that lead to a submitted review has been decreasing steadily over 13 years (2003–2015) for four of the six journals examined, with a cumulative effect that has been quite substantial". Their data is stored in `../data/FoxEtAl.csv`. We're going to build models trying to predict whether a reviewer will agree (or not) to review a manuscript.

```
# read the data
reviews <- read.csv("../data/FoxEtAl.csv", sep = "\t")
# take a peek
head(reviews)
# set NAs to 0
reviews[is.na(reviews)] <- 0
# how big is the data?
dim(reviews)
# that's a lot! Let's take 5000 review invitations for our explorations;
# we will fit the whole data set later
set.seed(101)
small <- reviews[order(runif(nrow(reviews))),][1:5000,]
```

The response variable of interest is a reviewer i agreeing to review a manuscript or not, and is binary; and so we decided to use a logistic regression. Call π_i the probability that a reviewer i agree to review a manuscript. We model $logit(\pi_i) = \log(\pi_i/(1-\pi_i))$ as a linear function.

Constant rate

As a null model we build a model in which the probability of agreeomg to review does not change in time/for journals:

```
# suppose the rate at which reviewers agree is a constant
mean(small$ReviewerAgreed)
# fit a logistic regression
model_null <- glm(ReviewerAgreed~1, data = small, family = "binomial")
summary(model_null)
# interpretation:
exp(model_null$coefficients[1]) / (1 + exp(model_null$coefficients[1]))
```

Declining trend

We now build a model in which the probability to review declines steadily from year to year:

```r
# Take 2003 as baseline
model_year <- glm(ReviewerAgreed~I(Year - 2003), data = small, family = "binomial")
#The I() function acts to convert the argument to "as.is"
summary(model_year)
```

Journal dependence

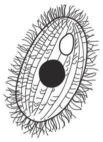
Reviewers might be more likely to agree for more prestigious journals:

```r
# Take the first journal as baseline
model_journal <- glm(ReviewerAgreed~Journal, data = small, family = "binomial")
summary(model_journal)
```

Model journal and year

Finally, we can build a model combining both features: we fit a parameter for each journal/year combination

```r
# Take the first journal as baseline, the colon mark stands for interaction term only.
model_journal_yr <- glm(ReviewerAgreed~Journal:I(Year-2003),
                        data = small, family = "binomial")
#summary(model_journal_yr)
```



T. thermophila

Likelihoods

In R, you can extract the log-likelihood from a model object calling the function `logLik`

```r
logLik(model_null)
logLik(model_year)
logLik(model_journal)
logLik(model_journal_yr)
```

Interpretation: because we're dealing with binary data, the likelihood is the probability of correctly predicting the agree/not agree for all the 5000 invitations considered. Therefore, the probability of guessing a (random) invitation correctly under the first model is:

```r
exp(as.numeric(logLik(model_null)) / 5000)
```

while the most complex model yields

```r
exp(as.numeric(logLik(model_journal_yr))/ 5000)
```

We didn't improve our guessing much by considering many parameters! This could be due to specific data points that are hard to predict, or mean that our explanatory variables are not sufficient to model our response variable.

AIC

We can also calculate AIC for logistic models. In `R` you can compute AIC using:

```r
AIC(model_null)
AIC(model_year)
AIC(model_journal)
AIC(model_journal_yr)
```

As you can see, the model `model_journal_yr` has the smallest AIC among all and is preferred here.

BIC

In a similar vein, BIC (Bayesian Information Criterion) uses a slightly different penalization: $BIC = \log(n)p - 2\log L(\theta|D, M)$, where n is the number of data points. Again, smaller values stand for "better" models:

```r
BIC(model_null)
BIC(model_year)
BIC(model_journal)
BIC(model_journal_yr)
```

Note that according to BIC, `model_year` is favored. As mentioned before, BIC would penalize a complex model more.

Cross validation

One very robust method to perform model selection, often used in machine learning, is cross-validation. The idea is simple: split the data in three parts: a small data set for exploring; a large set for fitting; a small set for testing (for example, 5%, 75%, 20%). You can use the first data set to explore freely and get inspired for a good model. The data will be then discarded. You use the largest data set for accurately fitting your model(s). Finally, you validate your model or select over competing models using the last data set.

Because you haven't used the test data for fitting, this should dramatically reduce the risk of overfitting. The downside of this is that we're wasting precious data. There are less expensive methods for cross validation, but if you have much data, or data are cheap, then cross-validation has the virtue of being fairly robust.

Let's try our hand at cross-validation. First, we split the data into three parts:

```
reviews$cv <- sample(1:3, nrow(reviews), prob = c(0.05, 0.75, 0.2), replace = TRUE)
dataexplore <- reviews[reviews$cv == 1,]
datafit <- reviews[reviews$cv == 2,]
datatest <- reviews[reviews$cv == 3,]
# We've already done our exploration.
# Let's fit the data using model_journal
# and model_journal_yr, which seem to be the most promising
cv_model1 <- glm(ReviewerAgreed~I(Year-2003), data = datafit, family = "binomial")
cv_model2 <- glm(ReviewerAgreed~Journal:I(Year-2003), data = datafit,
      family = "binomial")
```

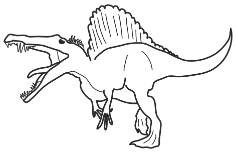
Now that we've fitted the models, we can use the function `predict` to find the fitted values for the `testdata`:

```
mymodel <- cv_model1
# compute probabilities
pi <- predict(mymodel, newdata = datatest, type = "resp")
# compute log likelihood
mylogLik <- sum(datatest$ReviewerAgreed * log(pi) +
                  (1 - datatest$ReviewerAgreed) * log(1 - pi))
print(mylogLik)
```

repeat for the other model

```
mymodel <- cv_model2
# compute probabilities
pi <- predict(mymodel, newdata = datatest, type = "resp")
# compute log likelihood
mylogLik <- sum(datatest$ReviewerAgreed * log(pi) +
                  (1 - datatest$ReviewerAgreed) * log(1 - pi))
print(mylogLik)
```

Cross validation supports the choice of the more complex model here.



S. aegyptiacus

Other approaches

Bayesian models are gaining much traction in biology. The advantage of these models is that you can get a posterior distribution for the parameter values, reducing the need for p-values and AIC. The downside is that fitting these models is computationally much more expensive (you have to find a distribution of values instead of a single value).

There are three main ways to perform model selection in Bayesian models:

- **Reversible-jump MCMC** You build a Monte Carlo Markov Chain that is allowed to "jump" between models. You can choose a prior for the probability of being in each of the models; the posterior distribution gives you an estimate of how much the data supports each model. Upside: direct measure. Downside: difficult to implement in practice – you need to avoid being "trapped" in a model.
- **Bayes Factors** Ratio between the probability of two competing models. Can be computed analytically for simple models. Can also be interpreted as the average likelihood when parameters are chosen according to their prior (or posterior) distribution. Upside: straightforward interpretation — it follows from Bayes theorem; Downside: in most cases, one needs to approximate it; can be tricky to compute for complex models.
- **DIC** Similar to AIC and BIC, but using distributions instead of point estimates.

Another alternative paradigm for model selection is Minimum-Description Length. The spirit is that a model is a way to "compress" the data. Then you want to choose the model whose total description length (compressed data + description of the model) is minimized.

A word of caution

The "best" model you've selected could still be a terrible model (best among bad ones). Out-of-fit prediction (such as in the cross-validation above) can give you a sense of how well you're modeling the data.

When in doubt, remember the (in)famous paper in Nature by Tatem et al. 2004, which used some flavor of model selection to claim that, according to their linear regression, in the 2156 Olympics the fastest woman would run faster than the fastest man. One of the many memorable letters that ensued reads:

> Sir — A. J. Tatem and colleagues calculate that women may out-sprint men by the middle of the twenty-second century (Nature 431,525; 2004). They omit to mention, however, that (according to their analysis) a far more interesting race should occur in about 2636, when times of less than zero seconds will be recorded.
>
> In the intervening 600 years, the authors may wish to address the obvious challenges raised for both time-keeping and the teaching of basic statistics.
>
> Kenneth Rice

Prediction for population outside the current data is prophecy.

P. polytes

V. Programming Challenge

P-hacking COVID-19

To show firsthand how p-hacking and overfitting are possible, we want you to show how these practices can lead to completely nonsensical results.

You can download a complete list of data on COVID-19 (coronavirus) by Our World in Data (https://ourworldindata.org/coronavirus). The data is updated daily and contains the lastest publicly available data on COVID-19 by country and by date. The data report the total cases, new cases, total deaths, new deaths, and hospitalization data of 233 countries and regions. Note that you are not expected to analyze the entire data. You may choose one or a few countries, or select one or some dates for analysis or for comparison.

The challenge is to build an analysis pipeline that produces a "significant" p-value for a relationship between COVID-19 cases and another variable, where the relationship is non-sensical, cannot possibly be causal, or could be argued either way. You may even simulate a fake variable as your key variable of interest. Prepare an Rmarkdown document with the results. At the end of the document write a paragraph to explain your "findings". As if you were in a debate team, pick on a subjective conclusion, and "cherry-pick" partial data to support your claim. Provide a non-statistical explanation for your group's fake result, and/or critique your statistcal approach and why your group got an apparently significant p-value.

As an example, below on a particular date (02/26/2020), I found a positive relationship between handwashing facilities and new cases in Asia countries.

Some sample code:

```r
library(dplyr)
#read the Dataset sheet into “R”. The dataset will be called "data".
data <- read.csv("https://covid.ourworldindata.org/data/owid-covid-data.csv",
     na.strings = "",header=T)


res <- NULL
for (i in 1:length(unique(data$date))){
  data1 <- data[which((data$date==unique(data$date)[i])&(data$continent=="Asia")),]
  data1 <- data1 %>% select("iso_code","date","new_cases", "handwashing_facilities")
  if (sum(rowSums(!is.na(data1[,3:4]))==2)>=10){
  res <- rbind(res, c(unique(data$date)[i],
      cor.test(data1[,3],data1[,4])$estimate,
      cor.test(data1[,3],data1[,4])$p.value))
}}

res[which((as.numeric(res[,2])>0)& (as.numeric(res[,3])<=0.05)),]
```

Workshops

RNAseq Workshop → Mengjie Chen
Population Genetics Workshop → Jeremy Berg
Immunology Workshop → Aly Khan

RNA-seq data analysis workshop[*]

Mengjie Chen (with Kate Farris) *University of Chicago*

This tutorial is going to walk you through basic RNA-seq data analysis using R. We will touch upon topics including quality control, differential expression analysis and downstream functional analysis.

Introduction

Transcriptomics is the study of the complete set of transcripts within a cell, which aims to document all species of transcripts, specifically mRNAs (along with ncRNAs), and quantify the gene expression levels during a particular biological process, in a development stage or under a set of unique pathological conditions like cancer. RNA-seq is a sequencing technique which uses next-generation sequencing (NGS) to reveal the presence and quantity of RNA in a biological sample at a given moment, analyzing the continuously changing cellular transcriptome. With maturation of NGS technologies, RNA-seq has become the main assay for transcriptomics studies.

The basic steps of RNA-seq data analysis are: assessing read quality, pre-processing (trimming), alignment to reference genome or assembly, quantification (of expression levels), and downstream functional analysis. In this workshop, we will skip some pre-processing steps and assume reads haven been aligned to the reference genome.

Step 1: Counting reads in genes

We will examine 8 samples from the `airway` package, which are from the paper by Himes et al: "RNA-seq Transcriptome Profiling Identifies CRISPLD2 as a Glucocorticoid Responsive Gene that Modulates Cytokine Function in Airway Smooth Muscle Cells".

To install these packages you can use the code (or if you are compiling the document, remove the `eval=FALSE` from the chunk.)

```r
install.packages("devtools")
install.packages("pheatmap")
BiocManager::install(c("airway","Rsamtools","Rsubread","DESeq2","vsn",
                       "org.Hs.eg.db","GenomicFeatures", "clusterProfiler"))
```

This workshop will focus on a summarized version of an RNA-seq experiment: a count matrix, which has genes along the rows and samples along the columns. The values in the matrix are the number of reads which could be uniquely aligned to the exons of a given gene for a given sample. We will demonstrate how to build a count matrix for a subset of reads from an experiment, and then use a pre-made count matrix, to avoid having students download the multi-gigabyte BAM files containing the aligned reads. A new pipeline for building count matrices, which skips the

[*] This document is included as part of the 'RNA-seq data analysis' workshop packet for the BSD qBio Bootcamp, University of Chicago, 2021. **Current version**: August 26, 2021.

alignment step, is to use fast pseudoaligners such as Sailfish, Salmon and kallisto, followed by the tximport package. See the package vignette for more details. Here, we will continue with the read counting pipeline.

First, make variables for the different BAM files and GTF file. Use the sample.table to contruct the BAM file vector, so that the count matrix will be in the same order as the sample.table.

```
library(airway)
dir <- system.file("extdata", package="airway", mustWork=TRUE)
csv.file <- file.path(dir, "sample_table.csv")
sample.table <- read.csv(csv.file, row.names=1)
bam.files <- file.path(dir, paste0(sample.table$Run, "_subset.bam"))
gtf.file <- file.path(dir, "Homo_sapiens.GRCh37.75_subset.gtf")
```

Next we create an Rsamtools variable which wraps our BAM files, and create a transcript database from the GTF file. We can ignore the warning about matchCircularity. Finally, we make a GRangesList which contains the exons for each gene.

```
library(Rsamtools)
bam.list <- BamFileList(bam.files)
library(GenomicFeatures)
txdb <- makeTxDbFromGFF(gtf.file, format="gtf")
exons.by.gene <- exonsBy(txdb, by="gene")
```

The following code chunk creates a SummarizedExperiment containing the counts for the reads in each BAM file (columns) for each gene in exons.by.gene (the rows). We add the sample.table as column data. Remember, we know the order is correct, because the bam.list was constructed from a column of sample.table.

```
library(GenomicAlignments)
se <- summarizeOverlaps(exons.by.gene, bam.list,
                        mode="Union",
                        singleEnd=FALSE,
                        ignore.strand=TRUE,
                        fragments=TRUE)
colData(se) <- DataFrame(sample.table)
```

Exercise: Can you check what is in the object exons.by.gene? Can you check what is in the object se?

M. mulatta

Step 2: Visualizing sample-sample distances

We now load the full SummarizedExperiment object, counting reads over all the genes.

```
library(airway)
data(airway)
airway
colData(airway)
rowRanges(airway)
```

The counts matrix is stored in assay of a `SummarizedExperiment`.

```
head(assay(airway))
```

Note that, on the un-transformed scale, the high count genes have high variance. That is, in the following scatter plot, the points start out in a tight cone and then fan out toward the top right. This is a general property of counts generated from sampling processes, that the variance typically increases with the expected value. We will explore different scaling and transformations options below.

Exercise: Can you plot the first two columns of `assay(airway)`? What do you observe? Can you plot samples from two different cell lines? Can you plot samples from two different treatment groups? What do you observe?

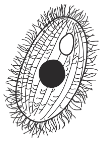
```
plot(assay(airway)[,1:2], cex=.1)
```

Step 3: Creating a `DESeqDataSet` object

We will use the `DESeq2` package to normalize the sample for sequencing depth. The `DESeqDataSet` object is just an extension of the `SummarizedExperiment` object, with a few changes. The matrix in assay is now accessed with counts and the elements of this matrix are required to be non-negative integers (0,1,2,...).

We need to specify an experimental design here, for later use in differential analysis. The design starts with the tilde symbol ~, which means, model the counts (log2 scale) using the following formula. Following the tilde, the variables are columns of the `colData`, and the + indicates that for differential expression analysis we want to compare levels of dex while controlling for the cell differences.

```
library(DESeq2)
dds <- DESeqDataSet(airway, design= ~ cell + dex)
head(dds)
```



T. thermophila

Step 4: Normalization for sequencing depth

The goal of normalization: to make RNA-seq samples comparable by taking into account differences in sequencing depths of RNA-seq libraries. Typical normalization methods include quantile normalization, median of ratios and trimmed mean of M values. Normalization method in DEseq2 is median of ratios, which counts for sequencing depth and RNA composition.

Size factors in DEseq2 are calculated by the median ratio of samples to a pseudo-sample (the geometric mean of all samples). In other words, for each sample, we take the exponent of the median of the log ratios in this histogram.

```r
loggeomeans <- rowMeans(log(counts(dds)))
hist(log(counts(dds)[,1]) - loggeomeans,
     col="grey", main="", xlab="", breaks=40)
```

The size factor for the first sample:

```r
exp(median((log(counts(dds)[,1]) - loggeomeans)[is.finite(loggeomeans)]))
```

```r
sizeFactors(dds)[1]
```

In DEseq2, a size factor will be estimated for each sample.

```r
dds <- estimateSizeFactors(dds)
sizeFactors(dds)
```

```r
colSums(counts(dds))
```

```r
plot(sizeFactors(dds), colSums(counts(dds)))
abline(lm(colSums(counts(dds)) ~ sizeFactors(dds) + 0))
```

Exercise: Is your calculated exponent of the median of the log ratios the same as output of function sizeFactors? Can you repeat the analysis for sample 5? Based on the above plot, what is the relationship between sizeFactors(dds) and column sum of count matrix? Are they equal? Why?

Make a matrix of log normalized counts (plus a pseudocount):

```r
log.norm.counts <- log2(counts(dds, normalized=TRUE) + 1)
```

Another way to make this matrix, and keep the sample and gene information is to use the function normTransform. The same matrix as above is stored in assay(log.norm).

```r
log.norm <- normTransform(dds)
```

Examine the log counts and the log normalized counts (plus a pseudocount).

```r
rs <- rowSums(counts(dds))
par(mfrow=c(1,2))
boxplot(log2(counts(dds)[rs > 0,] + 1)) # not normalized
boxplot(log.norm.counts[rs > 0,]) # normalized
```

Exercise: How do you think of the normalization results? Are you satisfied? What happened after normalization? Make a scatterplot of log normalized counts against each other. Did you note the fanning out of the points in the lower left corner, for points less than 2ˆ5=32? Would you concern about it? Can you repeat for Sample 3 vs. Sample 5?

Make a scatterplot of log normalized counts against each other.

```
plot(log.norm.counts[,1:2], cex=.1)
```

D. melanogaster

Step 5: Stabilizing count variance

We will use a sophisticated transformation to address the variance for low counts. It uses the variance model for count data to shrink together the log-transformed counts for genes with very low counts. For genes with medium and high counts, the `rlog` is very close to `log2`.

We use the argument `blind=FALSE` which means that the global dispersion trend should be estimated by considering the experimental design, but the design is not used for applying the transformation itself. See the `DESeq2` vignette for more details.

```
rld <- rlog(dds, blind=FALSE)
plot(assay(rld)[,1:2], cex=.1)
```

Another transformation for stabilizing variance in the DESeq2 package is `varianceStabilizingTransformation`. These two tranformations are similar, the rlog might perform a bit better when the size factors vary widely, and the `varianceStabilizingTransformation` is much faster when there are many samples.

```
vsd <- varianceStabilizingTransformation(dds, blind=FALSE)
plot(assay(vsd)[,1:2], cex=.1)
```

We can examine the standard deviation of rows over the mean for the log plus pseudocount and the rlog. Note that the genes with high variance for the log come from the genes with lowest mean. If these genes were included in a distance calculation, the high variance at the low count range might overwhelm the signal at the higher count range.

```
library(vsn)
meanSdPlot(log.norm.counts, ranks=FALSE)
```

Exercise: Can you compare the scatter plot of sample 1 and 2 before and after variance stablization transformation? Can you make a meanSdPlot for rlog and VST, respectively, and then compare those with untransformed data? Can you use the same ranges for Y axis? Does the transformation fix your previous problem on high variance of low counts?

Hints for changing the Y ranges.

```
  library("ggplot2")
  msd <- meanSdPlot(assay(vsd), ranks=FALSE)
  msd$gg + ggtitle("") + scale_y_continuous(limits = c(0, 1))
```

Next we will introduce two useful analyses for visualization of sample to sample differences/distance. The principal components (PCA) plot is a useful diagnostic for examining relationships between samples. In PCA, the high dimensional data are projected into a lower dimensional space (usually 2D), where the largest variability is retained.

```
plotPCA(log.norm, intgroup="dex")
```

In addition, we can plot a dendrogram based on hierarchical clustering on Euclidean distance matrix.

```
plot(hclust(dist(t(log.norm.counts))), labels=colData(dds)$dex)
```

Exercise: Can you make a PCA plot for rlog? Can you make a PCA plot for VST? How do you interpret PC1 and PC2 before and after variance Can you make a dendrogram for rlog and VST? Based on HC plot, can you comments on the effect of variance stablizing transformation?

Step 6: Differential gene expression

1) Modeling counts using a negative binomial distribution

We will now perform differential gene expression on the counts, to try to find genes in which the differences in expected counts across samples due to the condition of interest rises above the biological and technical variance we observe.

We will use an overdispersed Poisson distribution – called the negative binomial – to model the raw counts in the count matrix. The model will include the size factors into account to adjust for sequencing depth. The formula will look like:

$$K_{ij} \propto \mathrm{NB}(s_{ij}q_{ij}, \alpha_i)$$

where K_{ij} is a single raw count in our count table, s_{ij} is a size factor or more generally a normalization factor, q_{ij} is proportional to gene expression (what we want to model with our design variables), and α_i is a dispersion parameter.

For the negative binomial, the variance parameter is called disperison, and it links the mean value with the expected variance. The reason we see more dispersion than in a Poisson is mostly due to changes in the proportions of genes across biological replicates – which we would expect due to natural differences in gene expression.

```
par(mfrow=c(3,1))
n <- 10000
brks <- 0:400
hist(rpois(n,lambda=100),
```

```r
     main="Poisson / NB, disp=0",xlab="",breaks=brks,col="black")
hist(rnbinom(n,mu=100,size=1/.01),
     main="NB, disp = 0.01",xlab="",breaks=brks,col="black")
hist(rnbinom(n,mu=100,size=1/.1),
     main="NB, disp = 0.1",xlab="",breaks=brks,col="black")
```

The square root of the dispersion is the coefficient of variation – SD/mean – after subtracting the variance we expect due to Poisson sampling.

```r
disp <- 0.5
mu <- 100
v <- mu + disp * mu^2
sqrt(v)/mu
sqrt(v - mu)/mu
sqrt(disp)
```

A number of methods for assessing differential gene expression from RNA-seq counts use the negative binomial distribution to make probabilistic statements about the differences seen in an experiment. A few such methods are `edgeR`, `DESeq2`, and `DSS`. Other methods, such as `limma+voom` find other ways to explicitly model the mean of log counts and the observed variance of log counts.

`DESeq2` performs a similar step to `limma` in using the variance of all the genes to improve the variance estimate for each individual gene. In addition, `DESeq2` shrinks the unreliable fold changes from genes with low counts, which will be seen in the resulting MA-plot.

2) Experimental design and running DESeq2

Remember, we had created the `DESeqDataSet` object earlier using the following line of code (or alternatively using `DESeqDataSetFromMatrix`).

```r
dds <- DESeqDataSet(airway, design= ~ cell + dex)
```

First, we setup the design of the experiment, so that differences will be considered across time and protocol variables. We can read and if necessary reset the design using the following code.

```r
design(dds)
design(dds) <- ~ cell + dex
```

The last variable in the design is used by default for building results tables (although arguments to results can be used to customize the results table), and we make sure the "control" or "untreated" level is the first level, such that log fold changes will be treated over control, and not control over treated.

```r
dds$dex <- relevel(dds$dex, "untrt")
levels(dds$dex)
```

The following line runs the DESeq2 model. After this step, we can build a results table, which by default will compare the levels in the last variable in the design, so the dex treatment in our case:

```r
dds <- DESeq(dds)
res <- results(dds)
```

3) Examining results tables

```r
head(res)
table(res$padj < 0.1)
```

A summary of the results can be generated:

```r
summary(res)
```

For testing at a different threshold, we provide the alpha to results, so that the mean filtering is optimal for our new FDR threshold.

```r
res2 <- results(dds, alpha=0.05)
table(res2$padj < 0.05)
```

4) Visualizing results

The MA-plot provides a global view of the differential genes, with the log2 fold change on the y-axis over the mean of normalized counts:

```r
plotMA(res, ylim=c(-4,4))
```

We can also test against a different null hypothesis. For example, to test for genes which have fold change more than doubling or less than halving:

```r
res.thr <- results(dds, lfcThreshold=1)
plotMA(res.thr, ylim=c(-4,4))
```

A sorted results table:

```r
resSort <- res[order(res$padj),]
head(resSort)
```

Examine the counts for the top gene, sorting by p-value:

```r
plotCounts(dds, gene=which.min(res$padj), intgroup="dex")
```

A more sophisticated plot of counts:

```
library(ggplot2)
data <- plotCounts(dds, gene=which.min(res$padj), intgroup=c("dex","cell"), returnData=TRUE)
ggplot(data, aes(x=dex, y=count, col=cell)) +
  geom_point(position=position_jitter(width=.1,height=0)) +
  scale_y_log10()
```

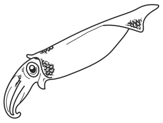
Connecting by lines shows the differences which are actually being tested by results given that our design includes cell + dex

```
ggplot(data, aes(x=dex, y=count, col=cell, group=cell)) +
  geom_point() + geom_line() + scale_y_log10()
```

A heatmap of the top genes:

```
library(pheatmap)
topgenes <- head(rownames(resSort),20)
mat <- assay(rld)[topgenes,]
mat <- mat - rowMeans(mat)
df <- as.data.frame(colData(dds)[,c("dex","cell")])
pheatmap(mat, annotation_col=df)
```

Exercise: Can you plot histogram for p-values? With what you have learned from previous lectures, is the histrogram with a desired shape? Can you make a heatmap based on top 50 genes? What do you observe?



D. pealeii

5) Getting alternate annotations

We can then check the annotation of these highly significant genes:

```
library(org.Hs.eg.db)
keytypes(org.Hs.eg.db)
anno <- select(org.Hs.eg.db, keys=topgenes,
               columns=c("SYMBOL","GENENAME"),
               keytype="ENSEMBL")
anno[match(topgenes, anno$ENSEMBL),]
```

Exercise: Based on the annotation, what is biological functions of top genes? Can you run a GO enrichment analysis using top 100 genes? You can use the following package or this website (http://geneontology.org/page/go-enrichment-analysis).

```
library(clusterProfiler)
top100genes <- head(rownames(resSort), 100)
annoList <- anno[match(top100genes, anno$ENSEMBL),]
ego <- enrichGO(gene          = annoList$ENSEMBL,
                universe      = rownames(resSort),
                OrgDb         = org.Hs.eg.db,
                keyType       = ’ENSEMBL’,
                ont           = "CC",
                pAdjustMethod = "BH",
                pvalueCutoff  = 0.01,
                qvalueCutoff  = 0.05,
                readable      = TRUE)
head(ego)
```

6) Looking up different results tables

The `contrast` argument allows users to specify what results table should be built. See the help and examples in `?results` for more details:

```
results(dds, contrast=c("cell","N061011","N080611"))
```

Exercise: Would you check the biological functions of top significantly differential expressed genes between cell line N061011 and N080611? Would you perform GO enrichment analysis on top genes as well? Any significant findings? Do you want to make any conclusion? Why or why not?

Hints:

```
res_contrast <- results(dds, contrast=c("cell","N061011","N61311"))
res_contrastSort <- res_contrast[order(res_contrast$padj),]
top100genes <- head(rownames(res_contrastSort), 100)
annoList <- anno[match(top100genes, anno$ENSEMBL),]
ego <- enrichGO(gene          = annoList$ENSEMBL,
                OrgDb         = org.Hs.eg.db,
                universe      = rownames(res_contrastSort),
                keyType       = ’ENSEMBL’,
                ont           = "CC",
                pAdjustMethod = "BH",
                pvalueCutoff  = 0.01,
                qvalueCutoff  = 0.05,
                readable      = TRUE)
head(ego)
```

Follow-up activities

`DESeq2` can be used to analyze time course experiments, for example to find those genes that react in a condition-specific manner over time, compared to a set of baseline samples. Here we demonstrate a basic time course analysis with the fission data package, which contains gene counts for an RNA-seq time course of fission yeast (Leong et al. 2014). The yeast were exposed to oxidative stress, and half of the samples contained a deletion of the gene `atf21`. We use a design formula that models the strain difference at time 0, the difference over time, and any strain-specific differences over time (the interaction term strain:minute).

```
library("fission")
data("fission")
ddsTC <- DESeqDataSet(fission, ~ strain + minute + strain:minute)
```

The following chunk of code performs a likelihood ratio test, where we remove the strain-specific differences over time. Genes with small p values from this test are those which at one or more time points after time 0 showed a strain-specific effect.

```
ddsTC <- DESeq(ddsTC, test="LRT", reduced = ~ strain + minute)
```

Exercise: Can you visualize results from the above analysis? Can you perform functional analysis? Can you propose other tests on this dataset?

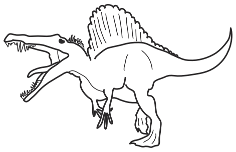
Acknowledgement

This workshop contains online materials written by Rafael Irizarry and Michael Love.

Session information

Here is the session information.

```
devtools::session_info()
```



S. aegyptiacus

Population genetics workshop[*]

Jeremy Berg *University of Chicago*

This exercise is going to expose you to several basic ideas in probability and statistics as well as show you the utility of using R for basic statistical analyses. We'll do so in the context of a basic population genetic analysis.

The scenario

As a biologist, you will learn what are the major patterns that are expected when the data you work with is clean. Using that expertise will save you from the mistake of misinterpreting error-prone data. In population genetics, there are a number of patterns that we expect to see immediately in our datasets. In this exercise you will explore one of those major patterns. Rather than give it away — let's begin some analysis and see what we find. In the narrative that follows, we'll refine our thinking as we go.

Introductory terminology

- Single-nucleotide polymorphism (SNP): A nucleotide base-pair that is *polymorphic* (i.e. it has multiple types or *alleles* in the population)
- Allele: A particular variant form of DNA (e.g. A particular SNP may have the "A-T" allele in one DNA copy and "C-G" in another; We typically define a reference strand of the DNA to read off of, and then denote the alleles according to the reference strand base - so for example, these might be called simply the "A" and "C" alleles. In many cases we don't care about the precise base, so we might call these simply the A_1 and A_2 alleles, or the A or a alleles, or the 0 and 1 alleles.)
- Minor allele: The allele that is more rare in a population
- Major allele: The allele that is more common in a population
- Genotype: The set of alleles carried by an individual (E.g. AA, AC, CC; or AA, Aa, aa; or 0, 1, 2)
- Genotyping array: A technology based on hybridization with probes and florensence that allows genotype calls to be made at 100s of thousands of SNPs per individual at an affordable cost.

The data-set and basic pre-processing

We will look at Illumina 650Y genotyping array data from the CEPH-Human Genome Diversity Panel. This sample is a global-scale sampling of human diversity with 52 populations in total.

The data were first described in Li et al (Science, 2008) and the raw files are available from the following link: http://hagsc.org/hgdp/files.html. These data have been used in numerous sub-

[*] Adapted from a workshop originally created by John Novembre. This document is included as part of the Population Genetics workshop packet for the BSD qBio Bootcamp, University of Chicago, 2022. **Current version**: August 11, 2022.

sequent publications (e.g Pickrell et al, Genome Research, 2009) and are an important reference set. A few technical details are that the genotypes were filtered with a GenCall score cutoff of 0.25 (a quality score generated by the basic genotype calling software). Individuals with a genotype call rate <98.5% were removed, with the logic being that if a sample has many missing genotypes it may be due to poor quality of the source DNA, and so none of the genotypes should be trusted. Beyond this, to prepare the data for the workshop, we have filtered down the individuals to a set of 938 unrelated individuals. (For those who are interested, the data are available as plink-formatted files `H938.bed` `H938.fam`, `H938.bim` from this link: http://bit.ly/1aluTln). We have also extracted the basic counts of three possible genotypes.

The files with these genotype frequencies are your starting points.

D. melanogaster

Note about logistics

We will use some functions from the `dplyr` and `ggplot2` and `reshape2` libraries so first let's load them:

```
library(dplyr)
library(ggplot2)
library(reshape2)
```

If you get an error message saying there is no package titled `dplyr`,`ggplot2`, or `reshape2` you may need to first run `install.packages("dplyr")`, `install.packages("ggplot2")`, or `install.packages("reshape2")` to install the appropriate package.

We will not be outputting files - but you may want to set your working directory to the `sandbox` sub-directory in case you want to output some files.

The `MBL_WorkshopJJB.Rmd` file has the R code that you can run. Code is provided for most steps, but some you will need to devise for yourselves to answer the questions that are part of the workshop narrative.

Initial view of the data

Read in the data table:

```
g_raw <- read.table("../data/H938_chr15.geno", header=TRUE)
```

It will be read in as a dataframe in R.

Then use the "head" command to see the beginning of the dataframe:

```
head(g_raw)
```

You should see that there are columns each with distinct names.

```
CHR        SNP A1 A2 nA1A1 nA1A2 nA2A2
```

- CHR: The chromosome number. In this case all SNPs are from chromosome 15.
- SNP: The rsid of a SNP is a unique identifier for a SNP and you can use the rsid to look up information about a SNP using online resource such as dbSNP or SNPedia.
- A1: The minor allele at the SNP.
- A2: The major allele.
- nA1A1 : The number of A1/A1 homozygotes.
- nA1A2 : The number of A1/A2 heterozygotes.
- nA2A2 : The number of A2/A2 homozygotes.

Calculate the number of observations at each locus

Next compute the total number of observations by summing each of the three possible genotypes. Here we use the `mutate` function from the `dplyr` library to do the addition and add a new column to the dataframe in one nice step. Note: You could also subset the `nA1A1`, `nA1A2` and `nA2A2` columns and then use the `rowSums` function from the base R library. If you have time, you could try both versions and check that they give the same result.

```
g_raw <- mutate(g_raw, nObs = nA1A1 + nA1A2 + nA2A2)
```

Run `head(g)` and confirm your dataframe g has a new column called `nObs`.

Now use the `summary` function to print a simple summary of the distribution:

```
summary(g_raw$nObs)
```

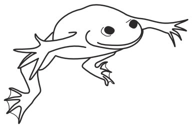
The `ggplot2` library has the ability to make "quick plots" with the command `qplot`. If we pass it a single column it will make a histogram of the data for that column. Let's try it:

```
qplot(nObs, data = g_raw)
```

Our data are from 938 individuals. When the counts are less than this total, it's because some individuals had array data that was difficult to call a genotype for and so no genotype was reported.

Question: Do most of the SNPs have complete data?
Question: Why might some SNPs have incomplete data?
Question: What proportion of SNPs have a missingness rate greater than 1.5%? Eliminate these SNPs from the dataset.

```
# Compute missingness rate
g_raw <- g_raw %>% mutate(missRate = 1  - (nObs / max(nObs)))

# Find the proportion of SNPs with missingness rate > 1.5%

# Filter based on missingness rate
g <- g_raw %>% filter(missRate < 0.015)
```



X. laevis

Calculating genotype and allele frequencies

Let's move on to calculating genotype and allele frequencies. For allele A_1 we will denote its frequency among all the samples as p_1, and likewise for A_2 we will use p_2.

```
# Compute genotype frequencies
g <- mutate(g, p11 = nA1A1/nObs , p12 = nA1A2/nObs, p22 = nA2A2/nObs )
# Compute allele frequencies from genotype frequencies
g <- mutate(g, p1 = p11 + 0.5*p12, p2 = p22 + 0.5*p12)
```

Question: With a partner or group member, discuss whether the equations in the code for p_1 and p_2 are correct and if so, why?

Run `head(g)` again and confirm that `g` now has the extra columns for the genotype and allele frequencies.

And let's plot the frequency of the major allele (A2) vs the frequency of the minor allele (A1). If we pass the `qplot` command two columns, it will plot them against one another. Let's try it here:

```
qplot(p1, p2, data=g)
```

Notice that $p_2 > p_1$ (be careful to inspect the axes labels here) This makes sense because A_1 is supposed to be the minor (less frequent) allele. Note also that there is a linear relationship between p_2 and p_1

Question: What is the equation describing this relationship?

The relationship exists because there are only two alleles - and so their proportions must sum to 1. The linear relationship you found exists because of this constraint. It also provides a nice check on our work (if p_1 and p_2 didn't sum to 1 it would suggest something is wrong with our code!).

Plotting genotype on allele frequencies

Let's look at an initial plot of genotype vs allele frequencies. We could use the base plotting functions, but the following uses the `ggplot2` commands. These are a little trickier, but end up being very compact (we need fewer lines of code overall to achieve our desired plot). To use `ggplot2` commands effectively our data need to be what statisticians call "tidy" (in this case, that means with one row per point we will plot).

To do this, we first subset the data on the columns we'd like (using the `select` command and listing the set of columns we want), then we pass this (using the %>% operator) to the `melt` command which will reformat the data for us, and output it as `gTidy`:

```
gTidy <- select(g, c(p1,p11,p12,p22)) %>% melt(id='p1',value.name="Genotype.Proportion")
head(gTidy)
ggplot(gTidy) + geom_point(aes(x = p1,
                               y = Genotype.Proportion,
                               color = variable,
                               shape = variable))
```

Now let's look at the graph that we produced. There is some scatter in the relationship between genotype proportion and allele frequency for any given genotype, but at the same time there is a very regular underlying relationship between these variables.

Question: What are approximate relationships between p_{11} vs p_1, p_{12} vs p_1, and p_{22} vs p_1? (Hint: These look like parabolas, which suggests are some very simple quadratic functions of p_1).

You might start to recognize that these are the classic relationships that are taught in introductory biology courses. If you recall, under assumptions that there is no mutation, no natural selection, infinite population size, no population substructure and no migration, then the genotype frequencies will take on a simple relationship with the allele frequencies. That is: $p_{11} = p_1^2$, $p_{12} = 2p_1(1-p_1)$ and $p_{22} = (1-p_1)^2$. In your basic texts, they typically use p and q for the frequencies of allele 1 and 2, and present these *Hardy-Weinberg proportions* as: p^2,$2pq$, and q^2.

Another way to think of the Hardy-Weinberg proportions is in the following way. If the state of an allele (A_1 vs A_2) is *independent* within a genotype, then the probability of a particular genotype state (such as A_1A_1) will be determined by taking the product of the alleles within it (so $p_{11} = p_1p_1$ or p_1^2).

Let's add to the plot lines that represent Hardy-Weinberg proportions:

```
ggplot(gTidy)+
  geom_point(aes(x=p1,y=Genotype.Proportion,color=variable,shape=variable))+
  stat_function(fun=function(p) p^2, geom="line", colour="red",size=2.5) +
  stat_function(fun=function(p) 2*p*(1-p), geom="line", colour="green",size=2.5) +
  stat_function(fun=function(p) (1-p)^2, geom="line", colour="blue",size=2.5)
```

On average, the data follow the classic theoretical expectations fairly well. It is pretty remarkable that such a simple theory has some bearing on reality!

By eye, we can see that the fit isn't perfect though. There is a systematic deficiency of heterozygotes and excess of homozygotes. Why?

Let's look at this more closely and more formally...

E. coli

Testing Hardy Weinberg

Pearson's χ^2-test is a basic statistical test that can be used to see if count data o_i conform to a particular expectation. It is based on the X^2-test statistic:

$$X^2 = \sum_i \frac{(o_i - e_i)^2}{e_i}$$

which follows a χ^2 distribution under the null hypothesis that the data are generated from a multinomial distribution with the expected counts given by e_i.

Here we compute the test statistic and obtain its associated p-value (using the `pchisq` function). We keep in mind that there is 1 degree of freedom (because we have 3 observations per SNP, but then they have to sum to a single total sample size, and we have to use the data once to get the estimated allele frequency, which reduces us down to 1 degree of freedom).

```
g <- mutate(g, X2 = (nA1A1-nObs*p1^2)^2 /(nObs*p1^2) +
              (nA1A2-nObs*2*p1*p2)^2 / (nObs*2*p1*p2) +
              (nA2A2-nObs*p2^2)^2 / (nObs*p2^2))
g <- mutate(g,pval = 1-pchisq(X2,1))
```

The problem of multiple testing

Let's look at the p-values for the first SNPs:

```
head(g$pval)
```

How should we interpret these? A p-value gives us the frequency at which the observed departure from expectations (or a more extreme departure) would occur if the null hypothesis that SNPs follows Hardy-Weinberg proportions is true. As an agreed upon standard (of the frequentist paradigm for statistical hypothesis testing), if the observation is relatively rare under the null (e.g. p-value < 5%), we reject the null hypothesis, and we would infer that the given SNP departs from Hardy-Weinberg expectations. This is problematic here though. The problem is that we are testing many, many SNPs (Use `dim(g)` to remind yourself how many rows/SNPs are in the dataset). Even if the null is universally true, we would expect to reject 5% of our SNPs using the standard frequentist paradigm. This is called the multiple testing problem. As an example, if we have 20,000 SNPs and the null hypothesis were true for all of them, on average we would naively reject the null for ~1000 SNPs based on the p-values < 0.05.

We clearly need some methods to deal with the "multiple testing problem". Two frameworks are the Bonferroni approach and false-discovery-rate (FDR) approaches. We will not say more about these here. Instead, we will do two simple checks to see though if our data are globally consistent with the null.

First, let's see how many tests have p-values less than 0.05. Is it much larger than the number we'd expect on average given the total number of SNPs and a 5% rate of rejection under the null?

```
sum(g$pval < 0.05, na.rm = TRUE)
```

Wow - we see many more. This is our first sign that although by eye these data show qualitative similarities to HW, statistically they are not fitting Hardy-Weinberg well enough.

Let's look at this another way. A classic result from Fisher is that under the null hypothesis the p-values of a well-designed test should be distributed uniformly between 0 and 1. What do we see here?

```
qplot(pval, data = g)
```

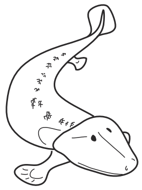
The data show an enrichment for small p-values relative to a uniform distribution. Notice how the whole distribution is shifted towards small values - The data appear to systematically depart from Hardy-Weinberg.

Plotting expected vs observed heterozygosity

To understand this more clearly, let's make a quick plot of the expected vs observed heterozygosity (the proportion of heterozygotes):

```
qplot(2*p1*(1-p1), p12, data = g) + geom_abline(intercept = 0,
                                                slope=1,
                                                color="red",
                                                size=1.5)
```

Most of the points fall below the y=x line. That is, we see a systematic deficiency of heterozygotes (and this implies a concordant excess of homozygotes). This general pattern is contributing to the departure from HW seen in the X^2 statistics.



T. roseae

Discussion: Population subdivision and departures from Hardy-Weinberg expectations

We might wonder why the departure from Hardy-Weinberg proportional is directional, in that, on average, we are seeing a deficiency of heterozygotes (and excess of homozygotes). One enlightening way to understand this is by thinking about what Sewall Wright (an eminent former University of Chicago professor) called "the correlation of uniting gametes". To produce an A_1A_1 individual we need an A_1-bearing sperm and an A_1-bearing egg to unite. If these events were independent of each other, we would expect A_1A_1 individuals at the rate predicted by multiplying probabilities, that is, p_1^2 (an idea we introduced above). However, what if uniting gametes are positively correlated, in that an A_1-bearing sperm is more likely to join with an A_1-bearing egg? In this case we will have more A_1A_1 individuals than predicted by p_1^2, and conversely fewer A_1A_2

individuals than predicted by $2p_1p_2$. If our population is structured somehow such that A_1 sperm are more likely to meet with A_1 eggs, then we will have such a positive correlation of uniting gametes, and the resulting excess of homozygotes and deficiency of heterozygotes.

Given the HGDP data is from 52 sub-populations from around the globe, and alleles have some some probability of clustering within populations, a good working hypothesis for the deficiency of heterozygotes in this dataset is the presence of some population structure.

While statistically significant, the population structure appears to be subtle in absolute terms — based on our plots, we have seen the genotype proportions are not wildly off from Hardy-Weinberg proportions.

Question: As an exercise, compute the average deficiency of heterozygotes relative to the expected proportion. This is the average of

$$\frac{2p_1(1-p_1)-p_{12}}{2p_1(1-p_1)}$$

What is this number for this data-set? A common "rule-of-thumb" for this deficiency in a global sample of humans is approximately 10%. Do you find this to be true from the data?

Just ~10% difference between expected and observed seems pretty remarkable given these samples are taken from across the globe. It is a reminder that human populations are not very deeply structured. Most of the alleles in the sample are globally widespread and not sufficiently geographically clustered to generate correlations among the uniting alleles. This is because all humans populations derived from an ancestral population in Africa around 100-150 thousand years ago, which is relatively small amount of time for variation across populations to accumulate.

Finding specific loci that show large departure from Hardy-Weinberg proportions

Now, let's ask if we can find any loci that are wild departures from Hardy-Weinberg proportions. These might be loci that have erroneous genotypes, or loci that cluster geographically in dramatic ways (such that they have few heterozygotes relative to expectations).

To find these loci, we'll compute the same relative deficiency you computed above, but let's look at it per SNP. This number is referred to as F by Sewall Wright and has connections directly to correlation coefficients (advanced exercise: Try to work this out!). If we assume there is no inbreeding within populations, this number is an estimator of F_{ST} (a quantity that appears often in population genetics as a measure of the degree of structure between populations).

Let's plot how it's value changes across the chromosome from one end to another:

```
qplot(y = Fstat, data = g, xlab = "SNP Number")
```

There are a few interesting SNPS that show either a very high or low F value.

Now, here's a trick. When a high or low F value is due to genotyping error, it likely only effects a single SNP. However, when there is some population genetic force acting on a region of the genome, it likely effects multiple SNPs in the region. So let's try to take a local average in a sliding window of SNPs across the genome, computing an average F over every 5 consecutive SNPs (in real data analysis we might use 100 kilobase or 0.1 centiMorgan windows).

The `stats::filter` command below calls the `filter` function from the `stats` library. The code instructs the function to take 5 values centered on a focal SNP, weighting them each by 1/5 and then taking the sum. In this way it produces a local average in a sliding window of 5 SNPs. Let's define the `movingavg` function and then make a plot of its values:

```
movingavg <- function(x, n=5){stats::filter(x, rep(1/n,n), sides = 2)}
avgF <- movingavg(g$Fstat)
qplot(x = seq(1, length(avgF)), y = avgF,  xlab = "SNP number", geom = "line")
```

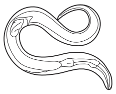
Wow — there appears to be one large spike where the average *F* is approximately 60% in the dataset!

Let's extract the SNP id for the largest value, and look at the dataframe:

```
outlier=which (movingavg(g$Fstat) == max(movingavg(g$Fstat),na.rm=TRUE))
outlier=which.max(avgF)
g[outlier,]
```

Question: Which SNP is returned? By inserting the rs id into the UCSC genome browser (https://genome.ucsc.edu/), and following the links, find out what gene this SNP resides near. The gene names should start with "SLC.." What gene is it?

Question: Carry out a literature search on this gene using the term "positive selection" and see what you find. It's thought the high *F* value observed here is because natural selection led to a geographic clustering of alleles in this gene region. Discuss with your partners why this might or might not make sense.



C. elegans

Discussion: The outlier region

The region you've found is one of the most differentiated between human populations that is known. Notice in your literature search, how it is known to affect skin pigmentation and is thought to contribute to differences in skin pigmentation that are seen between human populations. Finding strong population structure for alleles that affect external morphological phenotypes is not uncommon when looking at other chromosomes. Some of the most differentiated genes that exist in humans are those that involve morphological phenotypes - such as skin pigmentation, hair color/thickness, and eye color (the genes OCA2/HERC2, SCL45A2, KITLG, EDAR all come to mind). Many of these are thought to have arisen due to direct or indirect effects of adaptation to local selective pressures (e.g. adaptation to varying levels of UV exposure, local pathogens, local diets, local mating preferences), though in most cases we still do not yet have a fully convincing understanding of their evolutionary histories. Regardless of the reasons, it is notable that many of the features that humans see externally in each other (i.e. the morphological differences) are controlled by genes that are outliers in the genome. At most variant SNPs, the patterns of variation are much closer to those of a single random mating populations than they are at variant sites like EDAR. Put another way, a genomic perspective shows us many of the differences people see in each other are in a sense, just skin-deep.

Wrap-up

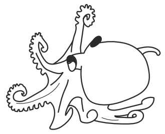
Modern population genetics has a lot of additional tools on its workbench, but here using relatively simple and classical ideas combined with genomic-scale data, we have been able to observe and interpret some major features of human genetic diversity. We have also revisited some basic concepts of probability and statistics such as independence vs correlation, the χ^2 test, and the problems of multiple testing. One remarkable thing we saw is that a very simple mathematical model based on assuming independence of alleles and genotypes can predict genotype proportions within ~10% of the true values. This gives us a hint of how simple mathematical models may be useful even in the face of biological complexity. Finally, we have gained more familiarity with R. We didn't discuss how genotyping errors might create Hardy-Weinberg departures, but if we were doing additional analyses, we could use Hardy-Weinberg departures to filter them from our data. It's common practice to do so, but with a Bonferonni correction and using data from within populations to do the filtering.

Follow-up activities

In the `data` folder, we are including data files that you can explore to gain more experience. These include global data for other chromosomes (`H938_chr*.geno`) and the same data but limited to different continental samples (`H938_*_chr*.geno`). Here are a few suggested follow-up activities. It may be wise to split the activities across class members and reconvene after carrying them out.

Follow-up activity: Look at chromosome 15 from your groups dataset - is the deficiency in heterozygosity as strong as it was on the global scale? Before you begin, what would you expect to see?

Follow-up activity Using your dataset, do you find any regions of the genome that are outliers for *F* on your chromosome? Using genome browsers and/or literature searches, can you find what is the likely locus under selection for that region? Be ready to share what you learned with the class.



O. bimaculoides

References

Li, Jun Z, Devin M Absher, Hua Tang, Audrey M Southwick, Amanda M Casto, Sohini Ramachandran, Howard M Cann, et al. 2008. "Worldwide Human Relationships Inferred from Genome-Wide Patterns of Variation." Science 319 (5866): 1100–1104.

Pickrell, Joseph K, Graham Coop, John Novembre, Sridhar Kudaravalli, Jun Z Li, Devin Absher, Balaji S Srinivasan, et al. 2009. "Signals of Recent Positive Selection in a Worldwide Sample of Human Populations." Genome Research 19 (5): 826–37.

Workshop on Computational Immunobiology

Aly A. Khan[1], Matt Schechter[2]
[1]Course Instructor; aakhan@uchicago.edu
[2]Course Assistant; mschechter@uchicago.edu

There's a joke about immunology, *which Jessica Metcalf of Princeton recently told me. An immunologist and a cardiologist are kidnapped. The kidnappers threaten to shoot one of them, but promise to spare whoever has made the greater contribution to humanity. The cardiologist says, "Well, I've identified drugs that have saved the lives of millions of people." Impressed, the kidnappers turn to the immunologist. "What have you done?" they ask. The immunologist says, "The thing is, the immune system is very complicated ... " And the cardiologist says, "Just shoot me now."* — Yong, Ed. "Immunology Is Where Intuition Goes to Die", The Atlantic, Aug. 2020.

Welcome. This workshop will introduce a few useful concepts for computationally thinking about the immune system. Immunology is currently undergoing a data-driven revolution, where advances in computer science and high-throughput technologies are allowing us to approach questions that previously could not be answered using purely experimental approaches or standard reductionist techniques. In this workshop we will explore two concepts:

BSD courses: Other courses that may broaden your experience in quantitative and computational immunology include: Microbial 'Omics (BIOS 25420) and Quantitative Immunobiology (IMMU 34800).

1. What are some methods for characterizing the phenotypic diversity and population structure of immune cells?
2. How can statistical modeling be used to make useful predictions about the immune system?

What is the immune system? The immune system can be viewed as a loosely connected network of cells that interact to solve problems that are beyond the individual capabilities of each cell. At the same time, the immune system has the ability to communicate between cells, coordinate collective action, and remember past events. The field of immunology has amassed a vast and specialized body of knowledge describing the common biological mechanisms underlying host defense, transplantation, autoimmunity, tumor immunology, allergy, and other clinical challenges.

The immune system's role in host defense is frequently framed by an attacked host (e.g., self) against alien invaders (e.g. non-self; viruses, bacteria, cancer cells, etc). This conflict has serious ramifications. Because collateral damage and misfiring can cause significant harm, the immune system must function precisely to distinguish between self and non-self (e.g. autoimmunity, allergy, etc). Strikingly, we still do not fully understand how the immune system accomplishes this exquisite role or why certain individuals develop autoimmunity.

We hope that as you progress through your graduate studies, you will notice that there are a myriad of exciting opportunities available to biologists who are willing to cross and explore the traditional boundaries between immunology and computer science.

1. Visualizing the population structure of immune cells

This section is adapted from Neu et al., Trends in Immunology, 2017.

In the past decade advances in computing and next-generation sequencing technologies have ushered in a new era of discovery in immunology. In particular, single cell RNA-seq (scRNA-seq) has enabled an unprecedented view of gene expression in single cells. A key challenge lies in visualizing single cell gene expression data in a biologically meaningful way while remaining robust to the high levels of noise that is present in single cell data.

scRNA-seq is RNA-seq performed on an individual cell. The cellular mRNA is amplified through oligos specific for the 5′ or 3′ tail of mRNA molecules or random hexamers.

One of the most compelling applications of single-cell genomics to immunology resides in characterizing the population structure of single cells. Visualization of scRNA-seq data can help to identify rare and intermediate subpopulations that are often overlooked with bulk RNA-seq data. The goal of visualization algorithms is to project high-dimensional data into a low-dimensional space, resolving cellular groups based on transcriptional similarity without the use of predetermined markers to determine their identity. In this section, we will look at two common dimensionality reduction algorithms that are used to visualize scRNA-seq data.

Dimensionality reduction is a process to reduce the number of variables to a compressed set of principal variables. More specifically, dimensionality reduction can be understood as projecting the data from the original high-dimensional space into much lower-dimensional space, while (roughly) capturing the concerned statistical properties (e.g., variation, distribution) and/or structure property (e.g., clusters). High-dimensional data after dimensionality reduction are easier to store and faster for downstream computation. Moreover, when projecting the data to two- or three-dimensional space, it is also easier for visualization.

Exercise 1.1

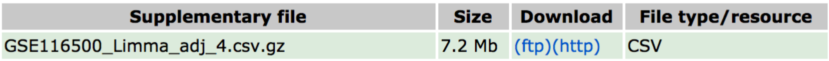
Exercise 1.1 — Overview discussion

Our goal for this exercise is to load and examine some real world scRNA-seq data. If you are new to scRNA-seq, let's take a moment to consider how and what format such data could be stored in. Let's also discuss what might be some prerequisite steps for analyzing it, such as quality control. We can also review concepts such as data sparsity, dimensionality reduction, and batch correction.

Exercise 1.1 — Data wrangling

Let's begin by downloading some scRNA-seq data. For this exercise we will be using data published as part of a study examining certain B cells in humans after influenza vaccination (Neu et al., JCI, 2019). We will download pre-processed supplementary data from the GEO database: `https://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE116500`. Scroll to the bottom of the page to locate the supplementary data.

GEO is a database managed by the NIH and functions as a public repository of high throughput sequencing and microarray data.



Supplementary file	Size	Download	File type/resource
GSE116500_Limma_adj_4.csv.gz	7.2 Mb	(ftp)(http)	CSV

Figure 1: The supplementary data can be found at the bottom of the page, and has has already been pre-processed.

The resulting csv.gz file contains single cell gene expression data for nearly 300 B cells. Let's load the data in R:

```r
# load data
geo <- read.csv('./data/GSE116500_Limma_adj_4.csv.gz',
    row.names = 1, header = TRUE)
```

How many genes and cells are in the data? Let's try looking at the dimensions of the data:

```r
# number of rows (Genes) and columns (cells)
dim(geo)
num_genes <- dim(geo)[1]
num_cells <- dim(geo)[2]
```

To made things easier for subsequent steps, let's take the transpose of the data so that each row denotes a cell and each column denotes a gene. The transpose of a matrix is an operation which flips a matrix over its diagonal; that is, it switches the row and column indices. This can facilitate certain types of linear algebraic operations and calculations which operate on columns by default.

```r
# current data with genes by cells
dim(geo)

# transpose data to cells by genes
geo <- t(geo)
dim(geo)
```

Our assumption is that most gene expression in scRNA-seq data contains random noise due to technical variation. We would like to focus our analyses on genes with high variability, which may have a biological basis. For the purposes of this exercise we simply calculate variance of gene expression as a way to rank genes, but we note other methods (e.g. coefficient of variation) can be used as well.

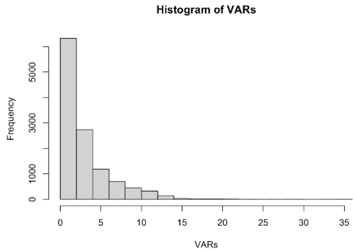


Figure 2: Histogram of gene expression variances.

```r
# Let's identify highly variable genes based on
# variance.
VARs <- apply(geo, 2, var)

# What is the distribution of variances?
```

```
hist(VARs)
```

Exercise 1.1 — Results discussion

What does our analysis of gene expression variance suggest to us about what is changing inside these cells? How should this observation inform our downstream tasks, such as data visualization?

Exercise 1.2

Exercise 1.2 — Overview discussion

Our goal for this exercise is to visualize our scRNA-seq data. Taking into account our observations about gene expression variances, how might we want to filter or preprocess our data? What is our intuition about good low-dimensional projections? Also, how can we evaluate the quality of our visualization?

Exercise 1.2 — Data Visualization

As we discussed earlier, we would like to focus our analyses on genes with high variability, which may have a biological basis. Let's select some top highly variable genes.

```
# we usually pick the top 1000 - 2000 highly
#   variable genes (HVGs)
hvgs <- names(sort(VARs, decreasing = TRUE))[1:2000]
geo.hvg <- as.data.frame(geo[,hvgs])
dim(geo.hvg)
```

We should now have a matrix of 295 cells with the top 2000 most highly variable genes. One fact that we did not reveal earlier is that these cells are plasmablasts, which are B cells that secrete antibodies. We will now classify the type (or isotype) of antibodies these B cells are secreting by comparing the expression of IgA and IgG genes. In order to help qualitatively examine various types of dimensionality reduction techniques, we

Immunoglobulin isotype can be thought of as a molecular classification for antibodies.

will classify and label each cell with their most highly expressed isotype:

```
# There are multiple genes that encode subclasses
# of the two isotypes in this data: IgA and IgG

# Grab max IgA values across all genes
IgA <- cbind(geo.hvg$IGHA1,geo.hvg$IGHA2)
IgA_max <- as.matrix(apply( IgA, 1, max))
#Set column name to 'IgA'
colnames(IgA_max) <- 'IgA'

# Grab max IgG values across all genes
IgG <- cbind(geo.hvg$IGHG1,geo.hvg$IGHG2,
    geo.hvg$IGHG3,geo.hvg$IGHG4)
IgG_max <- as.matrix(apply( IgG, 1, max))
#Set column name to 'IgG'
colnames(IgG_max) <- 'IgG'

#Determine if IgA is higher or IgG is higher
Ig <- cbind(IgA_max,IgG_max)
Ig_max <- colnames(Ig)[(apply( Ig, 1, which.max))]
```

We have now classified cells as IgA or IgG expressing plasmablasts.

Principal component analysis (PCA) is a linear dimensionality reduction algorithm that is often the first-step in visualizing high-dimensional data. We will need to use some functions from `ggplot2` in order to visualize our scRNA-seq data using PCA. Let's load the R package:

PCA is a linear dimensionality reduction algorithm, used to project high dimension data into a few 'components' that capture most of the variability in the data. It is a popular visualization technique that can help identify patterns or connections between samples.

```
library("ggplot2")
```

As a general rule of thumb, if you get an error message saying there is no package titled `ggplot2` you may need to first install the appropriate package:

```r
install.packages("ggplot2")
```

PCA takes an input of correlations between cells based on gene expression data, and identifies principal components corresponding to linear combinations of genes, which cumulatively capture the variability of the total dataset. When the data is projected against the first few components, which account for the largest amount of variation, distinct populations can be visually and biologically interpreted. Let's perform PCA:

```r
# Let's perform PCA
geo.pca <- prcomp(geo.hvg, center = TRUE,
    scale = TRUE)
plot_pc_data <- data.frame(
    PC1=geo.pca[["x"]][,'PC1'],
    PC2=geo.pca[["x"]][,'PC2'])
```

Let's use the top two principal components to visualize our data:

```r
#plot PCA results with Ig status
ggplot(plot_pc_data, aes(x=geo.pca[["x"]][,'PC1'],
    y=geo.pca[["x"]][,'PC2'], color=Ig_max)) +
    geom_point(shape=1) + theme_minimal() +
    geom_point(aes(color = Ig_max)) +
    theme(legend.position = "top")
```

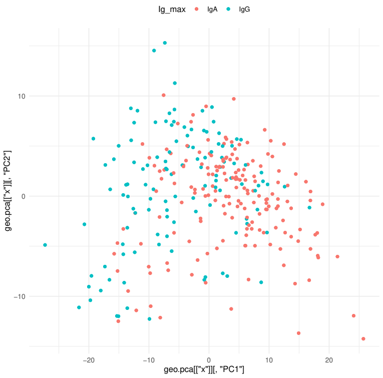


Figure 3: Visualizing our scRNA-seq data with PCA.

Do IgA cells and IgG cells separate well using PCA? Could we use other principal components?

T-DISTRIBUTED STOCHASTIC NEIGHBOR EMBEDDING (T-SNE) is a widely used nonlinear dimensionality reduction algorithm. Unlike PCA, which seeks to capture variance in data, t-SNE seeks to explicitly preserve the local structure of the original data. t-SNE constructs a probability distribution to describe the data set such that pairs of similar cells are assigned a high probability, while dissimilar pairs are assigned a much smaller probability. Thus, cells that are similar in the high-dimensional

t-SNE is a nonlinear dimensionality reduction method, which seeks to preserve the local structure of data in high-dimensional space when projected into low-dimensional space.

space will cluster together (due to high probability) in low-dimensional space. This ability to explicitly maintain clustering of similar cells is an advantage of t-SNE over direct linear transformation such as PCA. This approach is very effective with scRNA-seq data, and has been used to resolve transcriptionally distinct populations that are indistinguishable with PCA. We will need to use some functions from Rtsne, so let's load the R package:

```
library("Rtsne")
```

If you get an error message saying there is no package titled Rtsne you may need to first install the package.

```
# Let's use the top 10 PC for t-SNE
geo.tsne <- Rtsne(geo.hvg, theta = .001,
        perplexity = 30, initial_dims = 10)
plot_tsne_data <- data.frame(
    tsne1=geo.tsne$Y[,1],
    tsne2=geo.tsne$Y[,2])
```

Let's visualize our data:

```
#plot tsne results with Ig status
ggplot(plot_tsne_data, aes(x=geo.tsne$Y[,1],
    y=geo.tsne$Y[,2], color=Ig_max)) +
    geom_point(shape=1) + theme_minimal() +
    geom_point(aes(color = Ig_max)) +
    theme(legend.position = "top")
```

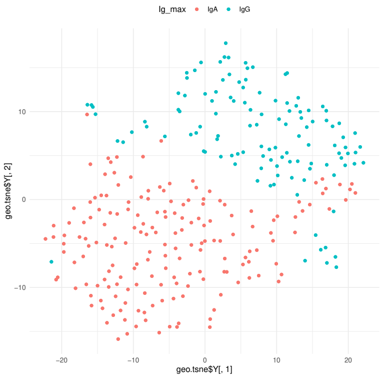


Figure 4: Visualizing our scRNA-seq data with t-SNE.

Do IgA cells and IgG cells separate well using t-SNE?

Exercise 1.2 — Results discussion

What does separability mean? What is "good" separability? Does it matter? How could we perform clustering on our visualizations? What would that tell us about the population structure of cells in the data? What might happen if we keep IgA and IgG as our cell labels but we remove expression information for all the underlying Ig genes and then visualize the data?

Conclusion

While single cell transcriptional profiles have high dimensionality due to the thousands of genes profiled, their intrinsic dimensionalities are typically much lower. Thus, unsupervised low dimensional projections can reveal salient structure in scRNA-seq datasets. However, the choice of dimensionality reduction algorithms used for visualization needs careful thought in immunology.

2. Peptide-MHC interactions

Nearly every cell in our body is decorated with a class of molecules known as major histocompatibility complex (MHC). The function of the MHC is to bind peptide fragments (antigens) derived from pathogens and display them on the cell surface for recognition by components of the immune system. In particular, alien peptides may be recognized by cytotoxic T cells, which can destroy infected cells. Understanding the binding affinity of MHC proteins and the repertoire of cognate peptide ligands is important for advancing our understanding of the antigenic landscape in infectious diseases, autoimmunity, vaccine design, and cancer immunotherapy.

Due to the importance of this process, peptide-MHC binding has been experimentally studied in a variety of ways. For example, the relative binding ability of different peptides to a specific MHC molecule can be directly assessed by competition experiments. The result of such experiments is a set of relative binding energies for different MHC-peptide combinations. In this section, we will model the repertoire of peptide ligands associated with a specific MHC molecule.

Exercise 2.1

Exercise 2.1 — Overview discussion

Our goal for this exercise is to load some HLA-peptide sequence and binding affinity data. If you are new to ana-

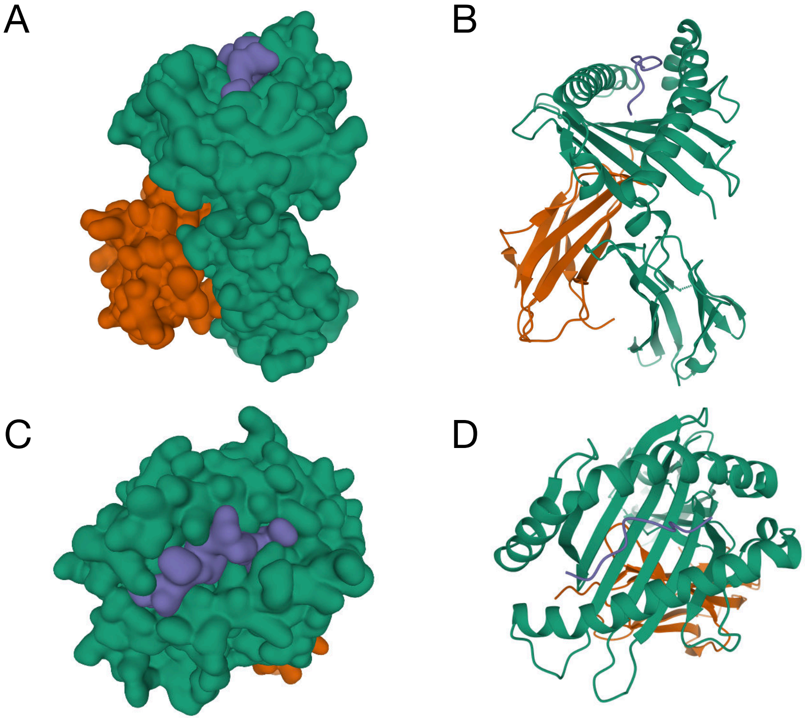


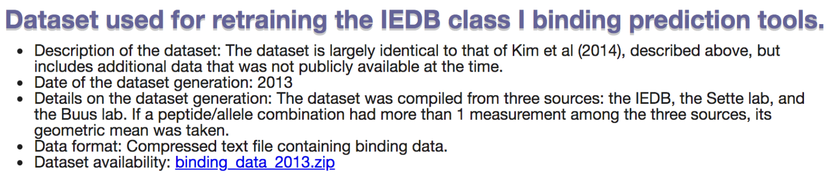
Figure 5: Structure of a peptide-MHC complex. (A) Front view of the crystal structure 1I4F depicting a space-filled molecular surface representation of the MHC (HLA-A*02:01) protein bound with a peptide (derived from MAGEA4, in purple), and (B) a backbone ribbon representation of the MHC. The heavy chain of MHC (alpha), which contains the binding cleft, is depicted in green. The supporting light chain (Beta-2 Microglobulin) is depicted in brown. (C,D) Top view of the complex showing the surface of the MHC (green) and the exposed surface of the bound peptide (purple). The exposed part of the peptide is referred to as the "TCR-interacting surface" of the peptide-MHC complex.

lyzing sequence data (or string data structures), let's take a moment to consider how this data structure differs from numerical data structures, and what are some operations we might want to perform on biological strings, such as sequence alignment. We can also review concepts such as edit distance and k-mers.

Exercise 2.1 — Data wrangling

High-throughout screening through competition experiments have resulted in large datasets cataloging binding affinities between various MHC molecules and peptides. Let's begin by downloading some peptide-MHC binding data. For this exercise we will be using data from the The Immune Epitope Database (IEDB) `http://tools.iedb.org/mhci/download/`
Let's download: `binding_data_2013.zip`
Once the peptide-MHC binding data is downloaded, you can uncompress the file. The resulting tab-delimited file

IEDB is a public database of immune epitope information. The database contains data related to antibody and T cell epitopes for humans, non-human primates, rodents, and other animal species. In particular, the database contains extensive MHC class I binding data from a variety of different antigenic sources.



Dataset used for retraining the IEDB class I binding prediction tools.

- Description of the dataset: The dataset is largely identical to that of Kim et al (2014), described above, but includes additional data that was not publicly available at the time.
- Date of the dataset generation: 2013
- Details on the dataset generation: The dataset was compiled from three sources: the IEDB, the Sette lab, and the Buus lab. If a peptide/allele combination had more than 1 measurement among the three sources, its geometric mean was taken.
- Data format: Compressed text file containing binding data.
- Dataset availability: binding_data_2013.zip

Figure 6: The binding data is available in the MHC class I section at the top (binding_data_2013.zip).

contains nearly 200,000 peptide-MHC combinations. Let's load the data using R:

```r
# load data
iedb <- read.csv('./bdata.20130222.mhci.txt',
    header = TRUE, sep = "\t", as.is =TRUE)

# let's use head to view a snippet of the data
head(iedb)
```

You should see the first few lines of the file, including the header for the columns. Let's take a moment to interpret what the values mean for each of the columns:

species This is the species from which a specific MHC allele was evaluated for peptide binding.

mhc This is the specific MHC allele.

peptide_length MHC class I molecules bind peptides that are predominantly 8-10 amino acid in length. Traditionally, there has been a focus on 9mer peptides when mapping HLA-I restricted T cell epitopes.

sequence This is the sequence of the peptide.

inequality This reflects the uncertainty for some of the peptide MHC binding data, where there some reported affinities are either an upper-bound or lower-bound to the true binding affinity.

meas The predicted output is given in units of IC50 nM. Therefore a lower number indicates higher affinity. As a rough guideline, peptides with IC50 values <50 nM are considered high affinity.

Exercise 2.1 — Results discussion

Why is this interesting? Several T-cell based cancer immunotherapies are being engineered to drive anti-tumor immune responses for specific antigens presented by the human MHC allele HLA-A*02:01. In cancer, somatic mutations altering the amino acid sequence of endogenous protein coding genes can result in the generation of tumor-specific HLA-presented antigenic peptide epitopes (or neo-antigens). These neo-antigens have the potential to activate cytotoxic T lymphocytes (e.g. CD8+ T cells) of the host immune system through HLA class I molecules, thereby provoking an anti-tumor immune response.

It stands to reason that if we knew the binding specificity of a given MHC, we could evaluate different somatic mutations in a cancer sample and determine if it could be presented by the cancer. Given the data available, can we model the repertoire of high affinity peptides that are presented by the human HLA-A*02:01 allele?

Exercise 2.2 — Overview discussion

Our goal for this exercise is to infer the pattern of amino acid specificity in high affinity peptides for a specific HLA. In other words, what positions and letters show a bias or preference for binding HLA-A*02:01? Let's assume each position of a peptide binds independently to the HLA molecule, how could we use intuition from statistics to approach this question? What are the possible letters or alphabet used in the peptides? How can we calculate what are the observed frequencies for a given position?

Exercise 2.2 — Data Visualization

One way to visualize the repertoire of high affinity peptides that can bind to HLA-A*02:01 is to use a sequence logo plot. First, the relative frequency of each amino acid at each position is calculated. This can be referred to as a positional weight matrix (PWM). Second, the logo plot depicts the relative frequency of each character by stacking characters on top of each other, with the height of each character proportional to its relative frequency. The total height of the letters depicts the information content of the position, in bits. Here, we will use an R package called `ggseqlogo` to calculate the position specific frequencies for all high affinity 9mer peptides and visualize the sequence logo.

A **PWM** is a type of scoring matrix in which amino acid substitution scores are inferred separately for each position from a collection of aligned protein sequences.

Notably, the Matthew Stephens Lab at UChicago has also developed a sequence logo tool called `Logolas`.

```r
# First let's install a seqlogo tool
install.packages("ggseqlogo")
library(ggseqlogo)

# You can also install Logolas
# BiocManager::install("Logolas")
# library(Logolas)
```

Let’s select peptides from our HLA of interest:

```r
# let's select human,  'HLA-A*02:01',
# peptides of length 9
# and binding affinity < 50
filtered_iedb = subset(iedb, species=='human'
    & mhc=='HLA-A*02:01'
    & meas < 50
    & peptide_length == 9)
```

Next, let’s try to model the distribution in a position specific manner:

```r
# let's grab the peptide sequences
listOfSequences = filtered_iedb[,4]

# number of sequences
numSequences = length(listOfSequences)

# length of each sequence
lengthOfSequence = nchar(listOfSequences[1])
```

```
# find unique characters in list of sequences
aminoAcidsVocab = unique(strsplit(
    paste(listOfSequences, collapse = ''),
    "")[[1]])

# create empty PFM matrix of zeros with dimensions
# of 20 x length of each sequence
PFM = matrix(0L, length(aminoAcidsVocab),
    lengthOfSequence)

# for loop through each sequence
for (sequence in listOfSequences) {

  # transform character vector to vector of
  # single characters # for looping
  sequenceString = strsplit(sequence, "")[[1]]

  # for loop through each amino acid in sequence
  for (index in seq_along(sequenceString)) {

    # increment value at PFM[amino acid
    # at position, positon]
    PFM[match(sequenceString[index],
        aminoAcidsVocab), index] =
    PFM[match(sequenceString[index],
        aminoAcidsVocab), index] + 1
    }
  }

# give rownames with amino acid letter
rownames(PFM) = aminoAcidsVocab
```

Let's pass the list of 9mers to ggseqlogo to visualize sequence logo:

```
ggseqlogo(peptide_sequences, seq_type='aa',
    as.is =TRUE)
```

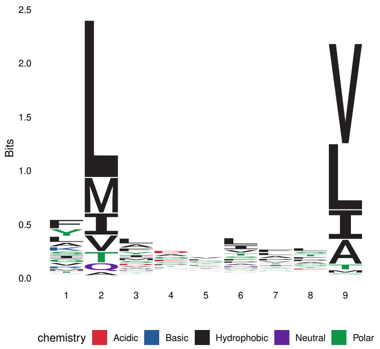


Figure 7: Sequence logo of high affinity peptides for HLA-A*02:01.

Exercise 2.2 — Results discussion

What positions of the high affinity peptides to seem to be highly specific for binding HLA-A*02:01? Let's use http://www.allelefrequencies.net to identify other HLA types. How does this compare with high affinity binding specificities of other HLA, such as HLA-C*06:02. What are the implications for minority populations?

Conclusion

Although we have made tremendous progress in modelling peptide-MHC interactions over the past several decades, connecting which T cells interact with which MHC-bound antigens remains a challenge. An efficent solution to this problem would have broad applications in improving our understanding of T cells in health, autoimmunity, and cancer (and potentially a free trip to Sweden). This remains a challenging task, in part, due to the enormous number of potential T cell receptors, the diversity of the MHC and bound antigen peptides. However, new computational methods may help resolve TCR-pMHC interactions by integrating and learning complex patterns from diverse high-throughput experimental approaches.

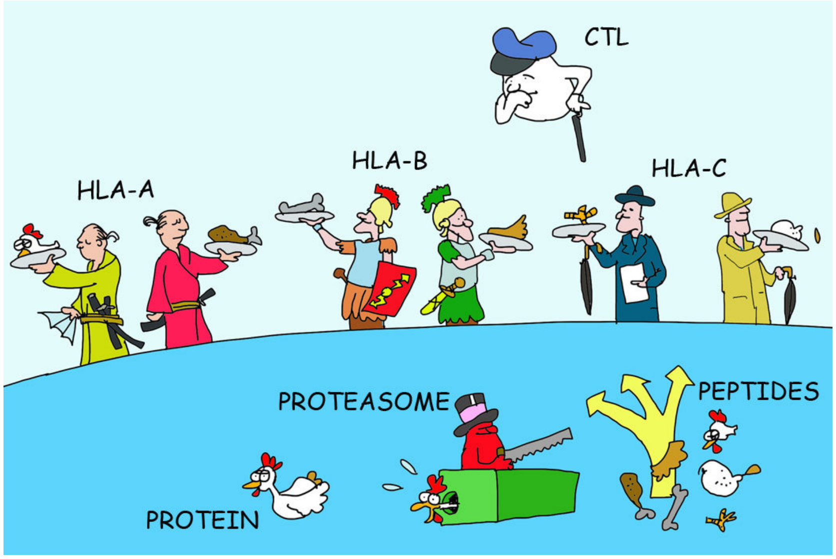


Figure 8: MHC class I overview (Rock, Kenneth L., Eric Reits, and Jacques Neefjes. "Present yourself! By MHC class I and MHC class II molecules." Trends in Immunology 37.11 (2016): 724-737.) https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5159193/figure/F1/