



ELEMENTS OF DATA SCIENCE AND STATISTICAL LEARNING

FALL 2017

OUTLINE

- Welcome to the class!
 - Organizational info: Course structure and format, schedule, expectations
- Introduction to R
 - What is R (and what it is not), and why?
 - Programming model, variables and data types
 - Vectors, matrices, vector and matrix indexing, vector arithmetic
- Quick review of basic statistical concepts:
 - Random variables, probability distribution and probability density function
 - Population vs sample, expected values and estimators; mean and variance
 - Hypothesis testing
 - Multiple variables: joint, marginal, conditional probability distributions

ABOUT THIS COURSE

- 15 weeks
 - 13 lectures – Wednesdays 7:40p; no lecture on mid-term week and final week; **NO CLASSES** on Nov. 22 (Thanksgiving)
 - Homework is assigned every week (total points per week are quiz+coding assignment):
 - An online Quiz in Canvas (no time limit): 40 points [meant to prompt review of lecture/book and earn points; please do not overthink them!]
 - Programming/data analysis assignment: 60 points [+ extra credit problems when offered]
 - Please check late submission policy and plan ahead – it is easy to fall behind with the coding assignments
 - Midterm and final exams: same format as programming homework assignments (no quizzes on those weeks):
 - Take home programming/data analysis assignment, with a full week to work on it; 100 points
 - Fewer specific instructions, more open exploration of the data – consider this a little bit of a *project*
 - You can be tested on *any* material learned prior to the exam
 - An on-campus section/lab/Q&A with instructors, Wednesday 6:35p (before the lecture)
 - Lectures and Wednesday on-campus sections are **streamed** and **recorded**.
- Communication with instructors and TFs: Piazza, email, in class and online (via zoom) Q&A sections (TBD)
- Questions, requests for clarifications, issues with HW and bugs in your programs, your own discoveries, ‘aha!’ moments, and useful resources you found elsewhere – please **ASK** and **SHARE**, respectively, using all the platforms available

WHAT TO EXPECT?

- This is an *introductory* course into statistical learning
 - We do not expect students to know stats beyond “classic” linear regression or to program beyond being very comfortable with writing a simple *working* program/script whenever needed (few dozen lines)
 - We are afraid (from experience) that at least some students might need to refresh their basic probability or for-loops.
 - Depending on how solid your pre-reqs are your mileage (and time spent on HWs) *will* vary
 - Being introductory, the course does not go too far or too deep into each of the topics studied, but there are **MANY** topics and statistical models discussed, and the course covers **LOTS** of ground.
 - You *will* learn enough to understand the fundamentals *and* to be able to apply all the discussed methods and models in practice, intelligently and meaningfully; you will be using real-life data throughout this course.
 - You will still have much to learn after we are done (don’t we all?!). Most of our weekly topics are whole research areas and could be a subject of a separate course.
 - We are trying our best at presenting a coherent, common “story” throughout the course – please try to see it. If different topics, methods, facts feel completely random and disjoint – something must be wrong; please revisit material, ask questions.
 - The course was found sufficiently high-paced by many students who took it
 - Plan ahead, start HWs early (allow yourself time to ask for help and act on suggestions received), ask questions, participate in discussions
 - The textbook is a classic. We are trying to *supplement* it as much as possible rather than just to retell. **READ** the text. **TRY** the labs and exercises (in class we are using different datasets too!)

OUTLINE

- Welcome to the class!
 - Organizational info: Course structure and format, schedule, expectations
- Introduction to R
 - What is R (and what it is not), and why?
 - Programming model, variables and data types
 - Vectors, matrices, vector and matrix indexing, vector arithmetic
- Quick review of basic statistical concepts:
 - Random variables, probability distribution and probability density function
 - Population vs sample, expected values and estimators; mean and variance
 - Hypothesis testing
 - Multiple variables: joint, marginal, conditional probability distributions

COMPUTATIONAL ENVIRONMENTS

- (Big) Data Science
 - Most statistical algorithms require extensive computation – many modern approaches would not probably even exist without computers
 - With the large amounts of data to be dealt with, even the simplest “classical” approaches can become unfeasible without computers
 - Software tools/libraries are required
- The usual trade-offs:
 - Simplicity, ease of use, flexibility, extensibility
 - Specifically designed tools: take one type of data, perform one type of analysis, generates results/plots: simple, easy to use (especially with a GUI), no flexibility, no extensibility. “I cannot do XYZ because my tool does not do it!”
 - General purpose programming language: *anything* can be written (specific programs from text editors through statistical software through computer games through device and equipment control software are all written in some language after all): ultimately flexible, extensible, not simple, not easy (may require LOTS of coding). “I cannot do XYZ because I am not a good programmer, or because it would take a month of work and 5000 lines of code”

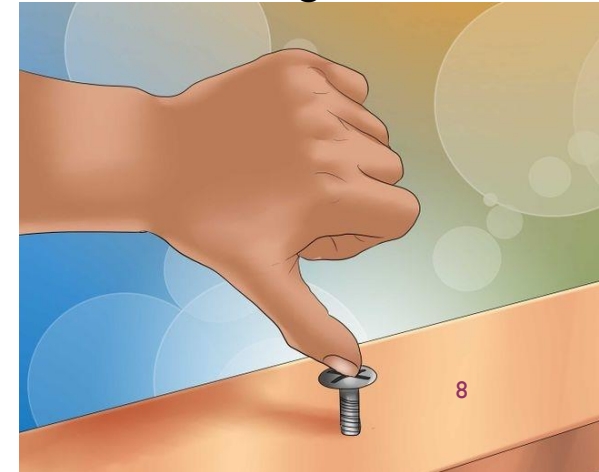
WHAT IS R?

- Lots of ground exists between the extremes: tools that provide a collection of predefined analyzes and maybe some extensibility (e.g. Excel), libraries/modules for general-purpose languages (e.g. C, Java, Python), etc.
- R is a framework that
 - Implements a full-featured programming language (called ‘S’)
 - Provides built-in relevant datatypes (numeric and character vectors, “tables”, etc), operations and facilities for data manipulation and analysis and for scientific visualizations
 - Includes (or allows seamless import of) large number of functions and modules (“packages”), in particular:
 - All sorts and flavors of statistical packages, both general purpose and domain-specific
 - Powerful plotting capabilities
 - Provides an interactive session, where commands can be executed one at a time and their outputs examined: one can really “play with the data”
 - Is free and widely supported: large number of books, online tutorials and discussion boards, lots of contributed packages from experts in different fields

IS R THE RIGHT TOOL?

- An analysis in R can be as simple as “load the data → call the right function → print/plot/examine the results”
 - Idealized picture, of course
- Is R the only tool? – NO
- Is R necessarily the best tool? – Questionable (depends on the job?)
- Does R provide a good overall balance between ease of use, power, flexibility and extensibility and should it be a top contender for almost any data analysis task? – Definitely!
 - Compare apples to apples: when we say “ease of use” for a platform that adopts a full-featured language, compare to other languages! Achieving the same results with a general purpose language would require lots of code and/or extensive custom libraries. And you may end up writing something similar to R!
 - Note: a popular Python module ‘pandas’ was modeled after R!

Use the right tool!



DRAWBACKS

- R is a simple language but is still a language: very complex tasks might be relatively easy to accomplish using built-in facilities, which is great, but even the simplest tasks still require writing some (small) chunk of code.
 - Example: no point and click interface. While in Excel you can create a simple graph by simply selecting, clicking and dragging, in R you will have to *describe* your plot by *typing commands*.
- Can be intimidating, one needs to get used to it:
 - A mix of the core language features, built-in functions, and extension libraries; the interfaces can be inconsistent (sometimes in very deceitful ways). Read documentation, test extensively!
 - R makes many (too many?) assumptions by design and tries its best (sometimes too hard) to execute the command it was given, even if this results in something very different from the programmer's intentions. Get into the habit of testing and validating intermediate steps!
 - Free – no professional/commercial support.

INSTALLING R

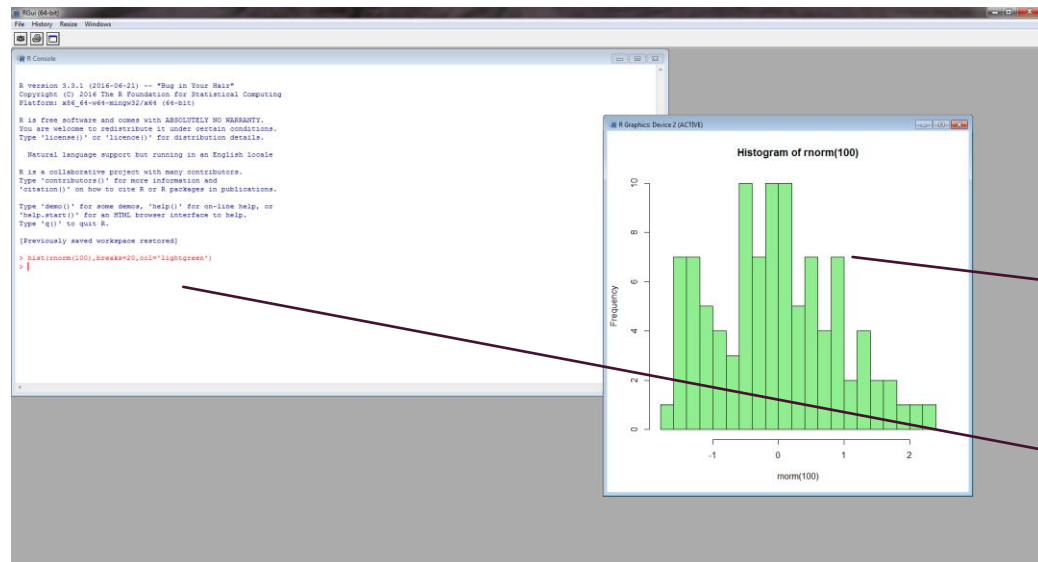
- R can be downloaded from CRAN archive: <http://cran.r-project.org>
 - Choose installer for your platform, download and run (for Windows, choose the 'base' package)
- Verify installation: click on the R icon on the desktop, run from the startup menu (Windows) or type R in the terminal window (Linux)
 - You will be presented with the 'command window', this is where you type the commands (on Linux, the terminal window itself will become R command window until you quit R).
- Type "help(help)" and press enter. You just called function help() (which displays documentation) with argument 'help' (i.e. you are asking to show usage documentation for the command (function) 'help' itself). **help() is your best friend!**
- Type quit() and press enter. This will end your R session and close the command window. If prompted whether you want to save session data, answer 'No'

INSTALLING R STUDIO

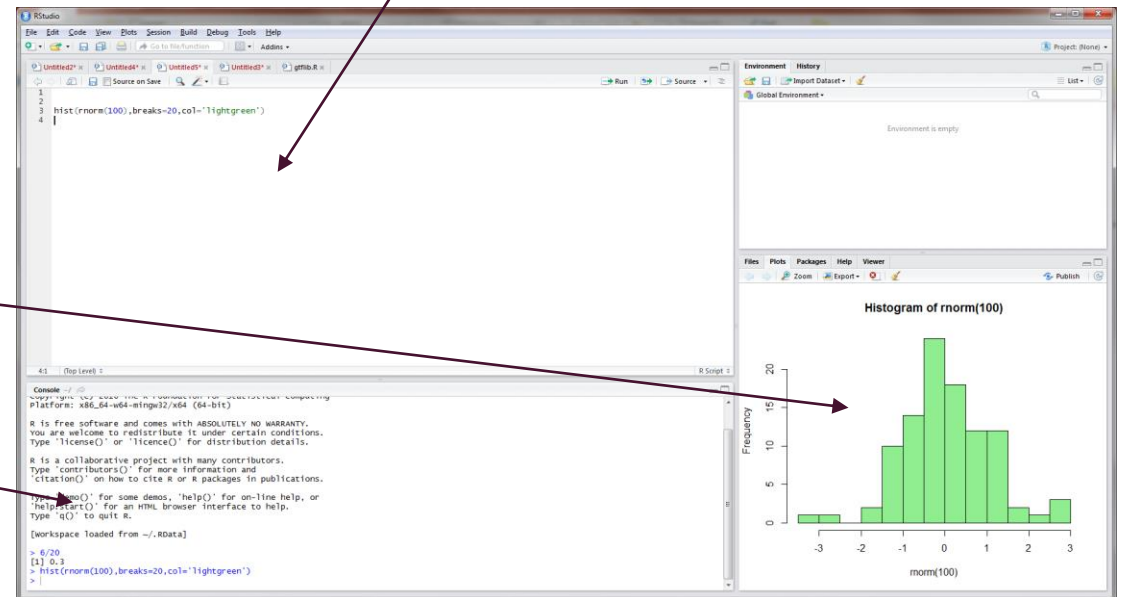
- In this course we ask you to use a GUI front-end called R Studio (it's actually great and it is a de facto industry standard for most intents and purposes)
 - Download and install: <https://www.rstudio.com/>
 - Note that you absolutely can run the interpreter/develop R code using 'bare' installation of R
 - In R studio, you will have the same R interpreter running underneath, integrated with code editor, plus many additional neat functions and capabilities...
 - ...such as “knitting HTML” – which you **need in order to submit the homeworks** (this capability can be installed into “bare” R as well, just easier with R studio as it's already there)
 - Your homework solution must include *both* Rmarkdown script (that combines R code performing necessary computations, presentation of results in numerical and graphical formats as well as the narrative you provide for them) *and* HTML report generated by “knitting” it
 - In our Q&A section video we show an elementary example of how to do it – also more details on this below

R VS R-STUDIO

- Command Interpreter, plots, code editor are easily available

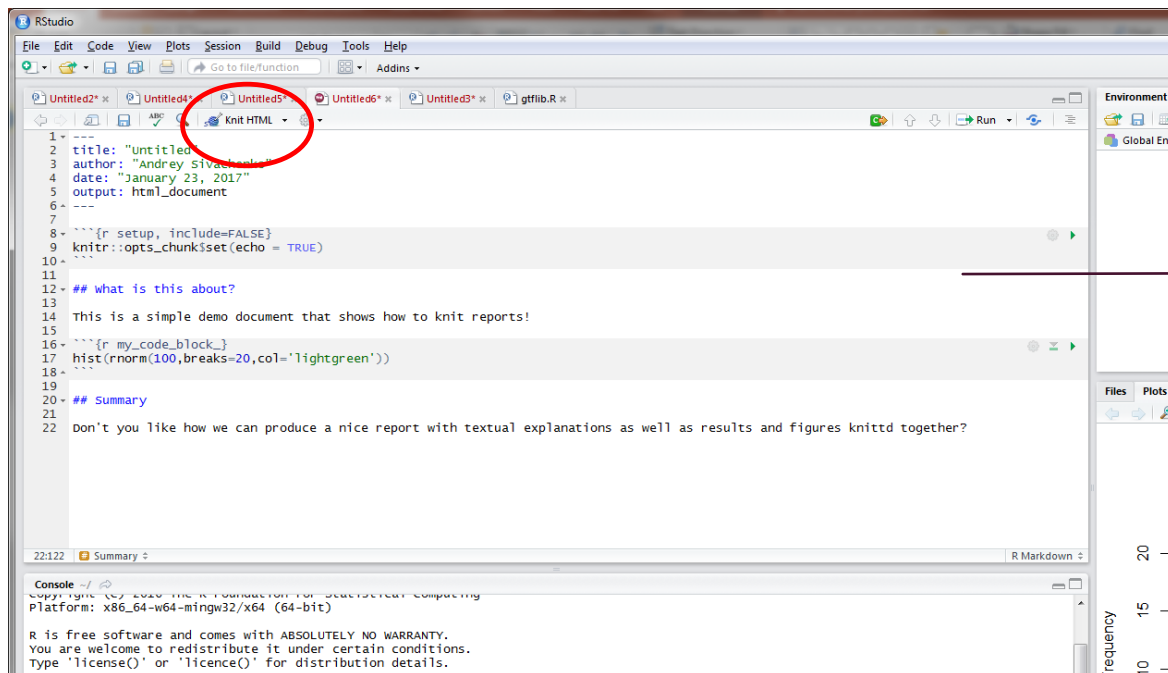


Code editor



KNITTING HTML DOCUMENTS

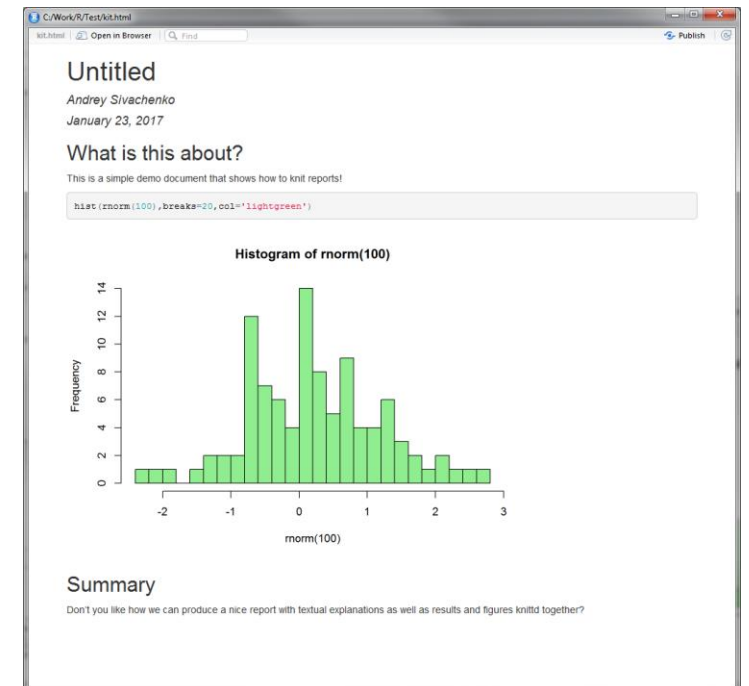
- A convenient approach that allows creating full-featured reports:
 - Actual R code (as well as some LaTeX-style formatting commands) can be embedded into the text document (.Rmd file)
 - As the document is processed (“knitted”) the code is executed and the results (numbers, tables, figures) are embedded into the output document (HTML or pdf), creating a report. Example:



The image shows the RStudio interface with an R Markdown file open. The code editor contains the following text:

```
1 ---  
2 title: "Untitled"  
3 author: "Andrey Sivachenko"  
4 date: "January 23, 2017"  
5 output: html_document  
6 ---  
7  
8 ## [r setup, include=FALSE]  
9 knitr::opts_chunk$set(echo = TRUE)  
10  
11  
12 ## What is this about?  
13  
14 This is a simple demo document that shows how to knit reports!  
15  
16 ## [r my_code_block]  
17 hist(rnorm(100,breaks=20,col='lightgreen'))  
18  
19  
20 ## Summary  
21  
22 Don't you like how we can produce a nice report with textual explanations as well as results and figures knitted together?
```

The "Knit HTML" button in the top toolbar is circled in red. An arrow labeled "knit" points from the RStudio window to the resulting HTML document on the right.



ON HOMEWORK SUBMISSIONS

- Expected submission format for programming/data analysis homework assignments: an Rmd file + knitted HTML
 - You can develop your code in interpreter/text editor (knitting is relatively slow and its not necessarily a good idea to knit while developing /debugging)
 - R Studio allows you to step through your code one statement at a time, execute just a selected code block from your Rmarkdown script, etc. – we show some examples of this in Q&A section video
 - When you are ready, copy your code into .Rmd (at the very least you can put all your code into a single code block; interspersing code with meaningful comments and explanations is of course infinitely better, that's what the Rmd format is for!); knit it, submit Rmd with your code AND the resulting HTML.
- On some rare occasions knitting may fail or even R Studio may crash (when the computation is too intense, no memory is left, etc). If you can still push your solution through using just the 'bare' R script file, you can submit just that file with the code PLUS all your results (numbers and figures as requested in the problem) in a separate pdf file (e.g. copy all your results/figures into a word document, save as pdf).
- None of the homework assignments for this course inherently require so much memory or CPU time that solutions for them cannot be knitted – key for every assignment will be posted in the form of R Markdown file and corresponding HTML output as soon as vast majority of the class has submitted their solutions

FIRST LOOK AT R

- In the following section we will look at some basic operations and functionality of R, specifically
 - Data types, variables, vectors and matrices
 - Arithmetic and logical expressions, vector indexing
 - Sequences and loops
 - Function calls
- **CHECKPOINT**: are you comfortable with these concepts, in *any* computer language?

YOUR FIRST R SESSION

- Start fresh R session. You will see a *command window* with the *command prompt* displayed in it
- R is an *interpreter*: a complete command is typed at the prompt (multiline input is allowed) and is executed *immediately*. The result is printed back into the command window and the new prompt is displayed (waiting for next command)
 - It is still possible to run a sequence of commands (an “R script”) at once, we will get to that later
 - It is possible to suppress command’s output and some commands are “silent”, but it’s still better to think in terms of “command→output” cycle

The diagram illustrates an R command window with a grey background. It shows a command prompt, a command being typed, a comment, and the output. Arrows point from text labels to the corresponding parts of the command window.

prompt

Command you type

Comment; you usually won't type comments in the interpreter mode, but it's a good idea to use comments in scripts. Examples will use comments too

```
> 2 + 2 # comment, ignored by R; command in this line adds 2 and 2
[1] 4
```

R executes the command and prints the output

R AS A CALCULATOR

- Just to get us started, try typing simple arithmetic expressions/math functions
 - Arithmetic expressions use the same syntax as in any other language
 - ...and so do functions (functionName(arg1,arg2,...))

```
> 72*84
[1] 6048
> (132+498+274)/23 # use parentheses to specify order of operations
[1] 39.30435
> sqrt(18.5) # square root
[1] 4.301163
> sin(72) # sine. Argument is assumed to be in radians
[1] 0.2538234
> sin(72*pi/180) # sine of 72 degrees (180 degrees equals pi radians)
[1] 0.9510565
```

VARIABLES

- Similarly to almost any other language, it is possible (and vital!) to keep the result of an expression or the value returned by a function call in a named variable
 - Variable names are case-sensitive, can use letters, digits, underscores and **DOTS**
 - “Traditional” assignment operator in R is “<-”, but “=” is also recognized
 - Variables can appear in expressions or as function arguments
 - Variables are *not declared* (assignment IS a declaration/definition): DYNAMIC typing

```
> x <- 2 # store value 2 in variable x
> y <- 3 # assign 3 to y
> x # a trivial expression
[1] 2
> y
[1] 3
> x+y # add values stored in x and y
[1] 5
```

```
> x*3 # continuing after the snippet on the left...
[1] 6
> x <- 4 # we can store new value in the same variable
> x*3 # multiplies current value of x by 3 (cf. above)
[1] 12
> z
Error: object 'z' not found
> z <- (x+y)*2 # can assign from any expression
> z # z is defined now
[1] 14
```

R DATA TYPES

- **CHECKPOINT:** What is a “data type” (as far as computer language definitions are concerned)?
- Typing system in R is STRONG
 - While we do not declare type of a variable (assignment IS a declaration/definition), *values* do have types
- Full definition of type in R is a little TRICKY
 - Almost of no importance for us, but good to remember/may help when you feel lost debugging an obscure error in your code
- There are mutually exclusive storage *modes* (describe how data are represented/stored in memory)
 - **Numeric:** 2, 1457, -675, 12.786, 6.022e23, ... – R does not make a clear distinction between integer and floating point numbers!
 - **Character:** "abc", "z", "this is a long string of text", '12.5', ... – either single or double quotes can be used (equivalent), no separate type for a single character (i.e. R’s “character” is what most other languages call a “string”)
 - **Logical:** TRUE, FALSE or, equivalently, T and F (shortcuts)
 - **List:** a compound/hierarchical type, a list of (arbitrary) values. Each value has its own mode.
 - Function, call, complex, ...

PRIMITIVE TYPES ARE VECTORS

- There is one significant difference with most other languages:
 - The chances are, any language you have worked with so far has primitive data types (same as we just discussed: integer/float, complex, character/string), which represent a single value (a “scalar”).
 - A different data type would be an *array* (or list) of primitive types (e.g. an array of integers)
 - In R, there is no such thing as a “scalar” value. The most primitive types are arrays (in R they are called “vectors”). A “single” value is just an array of length 1 – you can always add more elements!
 - One can create vectors of length > 1 “manually” (see below), also many functions can return vectors (as we will see later)

```
# note: commands in the same line should be separated
# with semicolon
> x="abc"; length(x)
[1] 1
> y <- 2:6; y # shortcut for seq(2,6,by=1) - see below
[1] 2 3 4 5 6
> z <- seq(1,1.4,by = 0.1); z # more general way
[1] 1.0 1.1 1.2 1.3 1.4
```

```
# c(concatenate) arbitrary vectors (of length 1
# in the following example) into a new vector:
> y.1=c(5,7,12,7); y.1
[1] 5 7 12 7
> y.2=rep(1:2,2); y.2 # repeat a vector
[1] 1 2 1 2
> c(y.1,y.2,-1) # concatenate again
[1] 5 7 12 7 1 2 1 2 -1
```

INDEXING A VECTOR (“BY POSITION”)

- Just like it is the case with arrays/lists in other languages, elements of vectors in R can be accessed individually
 - Indexing in R is way more powerful (except maybe for some dedicated numerical/scientific libraries, such as python’s Numpy)
 - **NOTE: vector indexes in R start from 1, not 0!**
 - Indexing operator is [] -- again, same as in most other languages

```
> clr<- c("red","blue","gray","pink","cyan"); clr
[1] "red" "blue" "gray" "pink" "cyan"
> clr[2] # so far so good, get the second element, any language can do that
[1] "blue"
> j=3; clr[j] # can use variables for indexing of course, the value in the var is the index
[1] "gray"
> clr[2:4] # can select a subset! (for those coming from Python: note the bounds of the range!)
[1] "blue" "gray" "pink"
# the indexes of the requested subset do not have to be continuous, ordered, or even unique.
# R will extract one element per each index specified:
> i=c(1,1,5,1); clr[i]
[1] "red" "red" "cyan" "red"
> clr[c(-2,-3)] # negative index specifies element to EXCLUDE
[1] "red" "pink" "cyan"
```

INDEXING A VECTOR (“BY FLAG”)

- Another indexing mode employs a vector of *logical* values as the index.
 - Since logical values are just TRUE/FALSE (think of them as “yes”/”no”, “on”/”off”), the indexing vector must have the same length as the vector being indexed: the elements at position(s) where indexing vector has TRUE are selected
 - Technically, you can use a shorter indexing vector, but it’s a shortcut: the indexing vector will be *recycled* automatically

```
> clr[c(T,F,F,T,F)] # select elements where index is T, i.e. 1st and 4th
[1] "red" "pink"
# select every other element; (T,F) is repeated until the target vector is exhausted:
> clr[c(T,F)]
[1] "red" "gray" "cyan"
> clr[T] # oooops, what happened here?
[1] "red" "blue" "gray" "pink" "cyan"
```

Checkpoint: make sure you understand the last example; consider the following scenario: in your code, you are *computing* the index of the element you would need to extract, you save that computed index in variable `i`, and then you execute `clr[i]`. Your calculation was supposed to give the value 1 for `i`. What if you accidentally got the result of the same calculation in logical mode?

COMPUTATIONS IN R ARE VECTOR-BASED

- Since primitive type is a vector, it is natural to have arithmetic operators, comparisons, and most functions *vectorized* too!
 - Arithmetic, comparisons, and logical operations are performed **element by element**
 - Assignments are also vector-based: if indexing expression is on the left hand side of the assignment, it selects subset of elements to assign rhs values to
 - In vector arithmetic operations, comparisons and assignments the operand is recycled if it's too short

```
> x<-c(3:8,NA) ; x
[1] 3 4 5 6 7 8 NA
> x > 5 # vector comparison (element by element), rhs recycling
[1] FALSE FALSE FALSE TRUE TRUE TRUE NA
> is.na(x) | x < seq(2,14,by=2) # combining two comparison results (element by element)
[1] FALSE FALSE TRUE TRUE TRUE TRUE TRUE
> x[x>5]=-1; x # NOTE: recycling of rhs occurred here; also note that x=-1 would override variable x
[1] 3 4 5 -1 -1 -1 NA
> x+c(1,3) # recycling occurs here too. Note the warning - only because "...length is not a multiple..." !
[1] 4 7 6 2 0 2 NA
Warning message:
In x + c(1, 3) :
  longer object length is not a multiple of shorter object length
```

USEFUL FUNCTIONS

- R offers huge number of built-in functions and even more are provided by extension packages (to be discussed later)
 - Here we review only few most common ones; we will see more as we progress

```
> x=seq(1,2,by=0.2) ; x
[1] 1.0 1.2 1.4 1.6 1.8 2.0

> length(x) # returns length of a vector
[1] 6

# math functions:
> exp(x)
[1] 2.718282 3.320117 4.055200 4.953032 6.049647 7.389056
> log(x) # natural logarithm
[1] 0.0000000 0.1823216 0.3364722 0.4700036 0.5877867 0.6931472
> sqrt(x)
[1] 1.000000 1.095445 1.183216 1.264911 1.341641 1.414214
# all the usual suspects: sin(), cos(), log2() and many more
# are also available
```

```
# sum of elements:
# generate a vector to play with
> xx <- c(-2:2,NA); xx
[1] -2 -1 0 1 2 NA
> sum(xx) # xx contains NA!
[1] NA
# same as sum(xx[!is.na(xx)]):
> sum(xx,na.rm=TRUE)
[1] 0
# can use sum in conjunction with a
# logical vector to count (remember,
# TRUE is 1 and FALSE is 0!):
> sum(xx>0)
[1] NA
> sum(xx>0,na.rm=T)
[1] 2
```


MATRICES

- Matrix is a two dimensional rectangular “table” of data
 - Just like vectors, matrices can only hold values of one type (mode), modes are the same as those for vectors
 - There are functions that return matrices, or one can use function `matrix()` to create one “manually”.
 - Function `matrix()` takes data values (can be recycled), number of rows and/or columns. By default, values fill column after column unless `byrow=TRUE` is specified (see `help()`). The values are always *stored* column after column (“column major order”)

```
> matrix(0,ncol=2,nrow=2)
      [,1] [,2]
[1,]    0    0
[2,]    0    0
> m <- matrix(1:6, nrow=2, byrow = T); m
      [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> matrix(11:16,nrow=2)
      [,1] [,2] [,3]
[1,]   11   13   15
[2,]   12   14   16
```

MATRICES: INDEXING

- Indexing operator takes **TWO** arguments, `m[row , column]`, but otherwise indexing for matrices and vectors is the same: can use vectors of positional indexes or logical flags for either rows or columns or both; negative index means skipping the corresponding row/column (again, similar to vectors)
 - Omitting row or column index vector altogether selects the whole row/column (but **DON'T FORGET THE COMMA!**)

```
> m <- matrix(1:12, nrow=3, byrow = T); m
      [,1] [,2] [,3] [,4]
[1,]  1    2    3    4
[2,]  5    6    7    8
[3,]  9   10   11   12
> m[1:2,3:4]
      [,1] [,2]
[1,]  3    4
[2,]  7    8
> m[1:2,c(4,3)]
      [,1] [,2]
[1,]  4    3
[2,]  8    7
```

```
> m[1:2,] # NOTE THE COMMA!
      [,1] [,2] [,3] [,4]
[1,]  1    2    3    4
[2,]  5    6    7    8
> m[1,]    # what has just happened???
[1] 1 2 3 4
# now we have a proper 1-row matrix
> m[1,,drop=F]
      [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
```

MATRICES: ASSIGNMENTS AND ARITHMETIC

- Other than needing TWO indexes (row and column), matrices behave very much like vectors:
 - Arithmetic, comparisons and logical operations are element by element (with *recycling* if necessary!)
 - Indexing on the lhs of the assignment causes assigning just to selected elements (and thus NOT reassigning new value to the variable itself)
 - **WARNING:** matrix can pose as a vector, in overwhelming majority of cases this is **NOT WHAT YOU INTEND**

```
> m <- matrix(1:12, nrow=3, byrow = T); m
     [,1] [,2] [,3] [,4]
[1,]  1   2   3   4
[2,]  5   6   7   8
[3,]  9  10  11  12
> m[1:2,2:3] + matrix(1:4,nrow=2)
     [,1] [,2]
[1,]  3   6
[2,]  8  11
> m[1:2,2:3] + c(3,5)
     [,1] [,2]
[1,]  5   6
[2,] 11  12
```

```
> m[1:2,2:3] + c(3,5,7)
     [,1] [,2]
[1,]    5  10
[2,]   11  10
Warning message:
In m[1:2, 2:3] + c(3, 5, 7) :
  longer object length is not a multiple of shorter... length
> m[5:7] # NOTE: NO COMMA!
[1] 6 10 3
> mode(m)
[1] "numeric"
> class(m) # !!!!! More on this later
[1] "matrix"
```

SOME USEFUL MATRIX FUNCTIONS

- These are very basic functions related to matrix size/structure. There is a large number of more “meaningful” functions that perform various types of calculations on matrices, we will encounter some of them later

```
> dim(m) # returns dimensions of a matrix (nrows, ncols)
[1] 3 4
> length(m) # What is length of a matrix?? Can you explain this??
[1] 12
> nrow(m) # how many rows?
[1] 3
> ncol(m) # how many cols?
[1] 4
> rownames(m) # rows of matrix m have no names so far...
NULL
> rownames(m) <- c('a','b','c') # assign names to rows
> colnames(m) <- c('w','x','y','z') # assign names to columns
> t(m) # transpose a matrix
```


	a	b	c
w	1	5	9
x	2	6	10
y	3	7	11
z	4	8	12

PROGRAMMING IN R: LOOPS

- Loops: for, while, repeat
- Loop control: `break` (break out of the loop immediately); `next` (jump to the head of the loop and start next iteration)

```
# for iterates loop variable over a vector:
> for ( i in 1:3 ) print(paste("iteration",i))
[1] "iteration 1"
[1] "iteration 2"
[1] "iteration 3"
> i # oooops, i is now defined!
[1] 3
# while loop is pretty standard: repeats while
# the loop condition is TRUE (must have length 1):
> i <- 1; while(i < 4) { print(i); i <- i+1 }
[1] 1
[1] 2
[1] 3
```

```
> x=c(1, NA, 5, -3, NA, 0)
> i = 0
> s = 0
> while( i < length(x) ) {
  i = i+1
  if ( is.na(x[i]) ) next
  s=s+x[i]
}
> s
[1] 3
```



- ! This is an example of a while/next AND of very inefficient code! Never use loops when you can use vector arithmetic instead! The same result as above could be obtained by `sum(x,na.rm=T)` or `sum(x[!is.na(x)])` !!

OUTLINE

- Welcome to the class!
 - Organizational info: Course structure and format, schedule, expectations
- Introduction to R
 - What is R (and what it is not), and why?
 - Programming model, variables and data types
 - Vectors, matrices, vector and matrix indexing, vector arithmetic
- Quick review of basic statistical concepts:
 - Random variables, probability distribution and probability density function
 - Population vs sample, expected values and estimators; mean and variance
 - Hypothesis testing
 - Multiple variables: joint, marginal, conditional probability distributions

RANDOM VARIABLES

- The notion of random variable is a cornerstone of statistical description of phenomena
- Random variable is a variable whose value is subject to variation due to random chance
 - Not the same as “variable” in a mathematical equation: $2x+4=10$ (unknown that can be calculated, at least in principle)
 - Think of random variable as a *process* that generates values (“measurements”, “observations”)
 - The next value is *not* known, but different values have different probabilities attached to them (we might or might not know those, or we can try *modeling* them)
 - Observed value(s) form a *realization* of a random process. Each realization is different from another.

EXAMPLES OF RANDOM VARIABLES

- A coin toss or a throw of a die. This is a process, which generates a value (head/tail or 1-6, respectively). The result of next toss is never known and can be characterized only by probability (coin=1/2, dice=1/6, if fair). This is an example of a **discrete** or **categorical** random variable (takes on a finite set of values).
- A subject is healthy or has a disease. Any given subject is either sick or healthy! But we know only when we “measure” in some way (e.g. run a diagnostic test) – it’s the same as observing heads or tails *after* the coin is flipped! We do not know if the *next randomly selected* person is sick or healthy, we may only know the probability (e.g. if disease prevalence=10%, the chance that the next random subject has a disease is $P=0.1$)
 - If we have *additional information* (some clinical parameters, e.g. body temperature, blood pressure, glucose levels etc) – only then we might be able to better “explain” the sick/healthy random variable, i.e. to make a better prediction/diagnosis. This is what this class is about!
- Body Mass Index: if we consider a population and draw subjects randomly, this is a random variable too! Also, price of a stock, price of a house, atmospheric pressure,
 - These are examples of **continuous** random variables

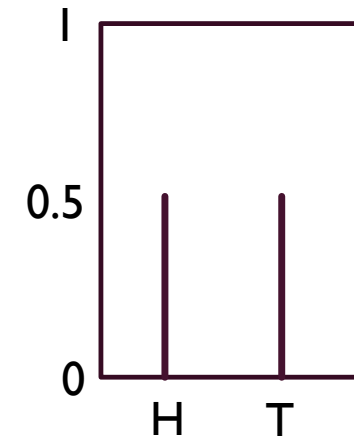
NOTATIONS

- We will (usually) denote a random variable with a capital letter, e.g. X
 - When considering multiple variables simultaneously we will use subscripts: X_1, X_2, \dots
- When the random variable takes specific value, we will denote this value with a small letter, e.g. x .
 - The fact that in a given trial the random variable X took value x (i.e. the observation) can be written as $X=x$
 - Note: this is not an equation or equality. Random variable is a *process* and thus it is much more than any given value it can take; this notation should be interpreted strictly in the sense described above.
 - For instance, if random variable X represents the process of tossing a coin, then if a trial (a toss) results in a head (h), we can write $X=h$, and if a different trial (for the same random variable!) results in a tail (t), we write $X=t$.

PROBABILITY DISTRIBUTION

- Discrete random variable X that can take any value from a finite set $\{x_0, \dots, x_n\}$ is *completely* characterized by probabilities of each value, $p(X=x_0)$, $p(X=x_1)$, \dots , $p(X=x_n)$. We may also denote these probabilities simply as p_0 , p_1 , \dots , p_n when it does not cause confusion
- Taken together, these probabilities form the *probability distribution* $P(X)$, which is a function of (a discrete set of) allowed values (aka probability mass function).
- Probability is normalized and the probability of a *certain* event is taken to be 1
 - Since one of the outcomes is bound to happen (e.g. we *will* observe either head or tail with certainty!), the probabilities of individual (disjoint) outcomes must sum to 1

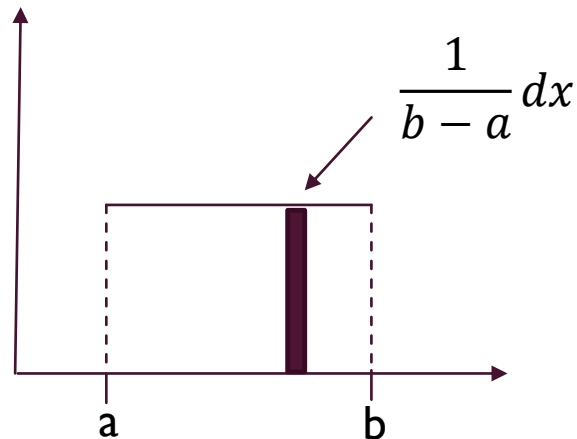
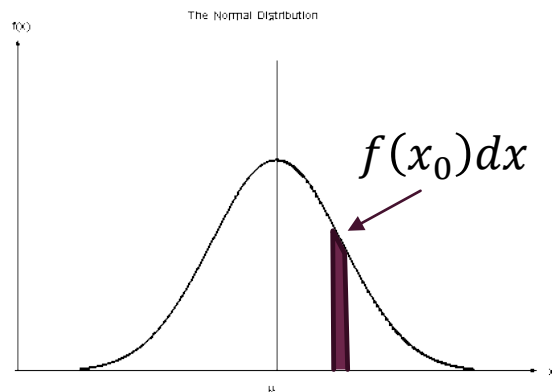
$$\sum p_i = 1$$



Probability distribution
of a fair coin

PROBABILITY DENSITY FUNCTION

- In the same way, a continuous random variable X is completely specified by the probabilities of each value it can take... in a sense
 - In contrast to discrete case, we cannot assign finite probability to any *exact* outcome value. What is the chance that upon randomly selecting a subject from the population the measurement of BMI will give exactly 22.439768778678976877762435...? Correct, it's 0.
 - Instead, for continuous random variable X we define a *probability density function* (PDF) $f(x)$ such that the probability to observe a value of the variable *within the small interval* dx around some x_0 is given by $P(x_0 - dx/2 < X < x_0 + dx/2) = f(x_0)dx$
 - The probabilities must be still normalized, $\int f(x)dx = 1$
 - The ubiquitous normal distribution (bottom left) is one example of a PDF. Another example shown at the bottom right is the *uniform* distribution (the probability to observe a value in a small interval of fixed length dx is *the same* across the whole region $[a,b]$)



MEAN AND VARIANCE

- The most important characteristics of any probability distribution/PDF are the *mean* and *variance*.

- **Mean** is simply the *expected value* of the random variable:

$$\mu = E[x] = \sum_i x_i p_i \text{ (discrete)}$$

$$\mu = E[x] = \int x f(x) dx \text{ (continuous)}$$

- **Variance** is the expected deviation from the mean, squared. It tells “how wide” the distribution is

$$\sigma^2 = \text{Var}(X) = E[(x - \mu)^2] = \sum_i (x_i - \mu)^2 p_i \text{ (discrete)}$$

$$\sigma^2 = \text{Var}(X) = E[(x - \mu)^2] = \int (x - \mu)^2 f(x) dx \text{ (continuous)}$$

- **Standard deviation** is simply the square root of the variance, by definition: $\sigma = \sqrt{\text{Var}(X)}$

- In general, mean and variance provide important information but *do not* characterize the distribution completely!

- Important exception: normal distribution $f(x|\mu, \sigma) = \frac{1}{\sqrt{2\pi} \sigma} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$

- We use the same letters as in the above definitions for a reason: if we calculate mean and variance of the normal, we will find that they are equal to the parameters μ, σ of the normal PDF, respectively

- As another example, the mean of the uniform distribution in the interval $[a,b]$ is $\mu=(a+b)/2$

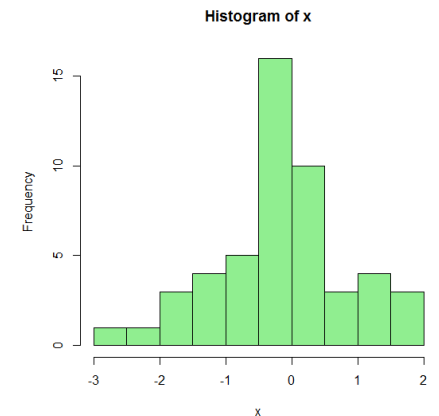
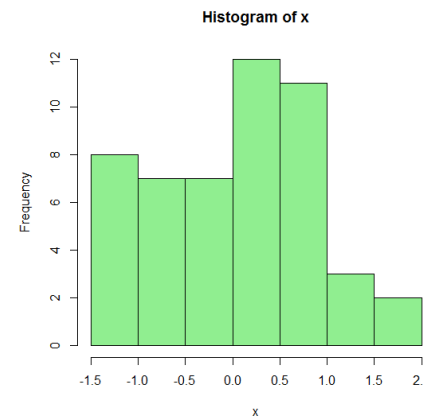
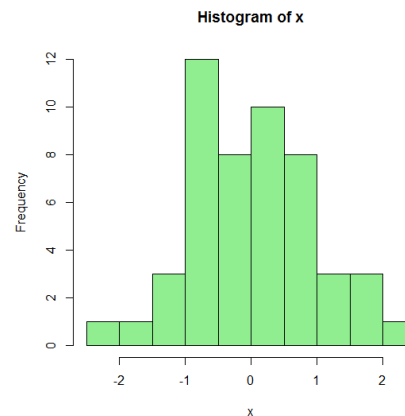
RANDOM SAMPLING

- The probability distribution or PDF determine the “true”, or “population” mean and variance
- Those are often not known, and in fact the purpose of the statistical analysis is often to determine them
- The experiment thus becomes:
 - A set of examples (a “sample”) is randomly drawn from the (unknown) underlying distribution
 - The population parameters (e.g. mean and variance) are inferred *from that sample*
- If we have a set of n measurements x_1, x_2, \dots, x_n (a sample) of the random variable X , then the *sample* mean is $\bar{\mu} = \frac{1}{n} \sum_{i=1}^n x_i$ - i.e. what you probably know as the “average”
 - Note that if we repeat the whole experiment again, i.e. draw a new sample of size n from the population, the observations x_i will be different in general (random variable!), hence a different average will be obtained
 - Thus the sample mean is itself a random variable that changes from sample to sample!
 - If the standard deviation of the random variable X is σ , then the standard deviation of the mean of a sample of n observations (also known as the standard error) is $\sigma_M = \sigma / \sqrt{n}$
- The standard deviation of a sample of size n is $s = \bar{\sigma} = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{\mu})^2}$ and is also a random variable
- The sample mean and sample variance/standard deviation are the *estimators* for the true parameters of the underlying distribution

SIMULATING RANDOM SAMPLING

- Let's use R to see how it works – you will need to build on this for this week homework assignment!
 - R has a collection of “samplers” for different distributions (see `help("Distributions")`). For instance, `rnorm(n, mean, sd)` returns a random sample of size `n` drawn from a normal distribution with the specified mean and sd (in other words the chances to get different observations will be governed by the requested normal)
 - The function returns sample as a vector

```
> x=rnorm(50,mean=0,sd=1)
> mean(x)
[1] 0.05167691
> hist(x,breaks=10,col='lightgreen')
```



- Repeating a few times, you will observe a new sample mean every time and the empirical distribution (the histogram) of the sample will also look differently every time (try it! – your results will also differ from run to run but will never be exactly the same as shown here!)
- Try much larger sample (e.g. $n=500$) and you will see the sample distribution that looks much closer to the actual (“underlying”) normal the sample was drawn from. As the sample size *increases* the sample estimators as well as the sample distribution as a whole approach the underlying distribution and its parameters
 - **A question to reflect upon:** if all we have is the sample of 50 measurements, as in the example above, how can we tell if the underlying distribution is normal? Can we at all?

HYPOTHESIS TESTING

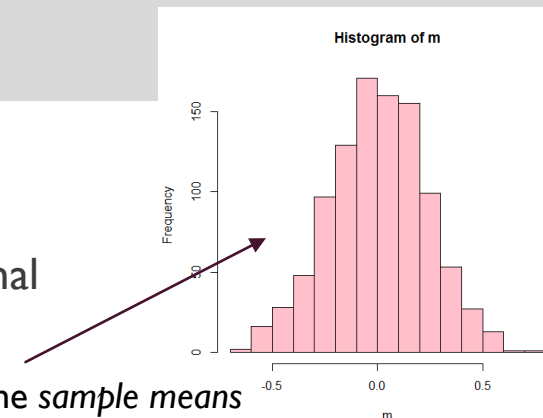
- Since we are dealing with random samples which have built-in error, the very meaningful (and in fact central) question is: when a certain effect (difference, trend, etc) is observed – *is it real?*
 - The question from the previous slide is just that type of question, in fact: is the *observed* (“empirical”) distribution “close enough” to (or “can be reasonably expected to be drawn from) underlying normal?
- Consider one of the simplest cases: a sample is drawn and its mean, μ is calculated.
 - It’s an *estimate* for the underlying population mean
 - Let’s say we could hypothesize that the underlying population mean could be 0
 - Calculated value is 0.23. **STOP and try to answer**: Does this mean that the underlying mean is *truly* non-zero? Does this mean it’s precisely 0.23? Somewhere in between?
 - The framework of hypothesis testing requires one to choose a *null hypothesis* (the one under which there is no true effect, generally), then calculate the probability to observe an effect/trend at least as extreme as the one actually observed, due solely to *randomness*. This probability is called *p-value*
 - Arguably, we can say that statistics is all about a single question: “*do we have enough evidence to believe that statement X is (un)true?*”

HYPOTHESIS TESTING SIMULATION

- Let us imagine that we observed 20 day-to-day changes in the stock price and the mean of this sample is 0.3, while sample variance is 1
- Null hypothesis: there is no trend, the distribution of the price changes has mean 0 and the observed change in mean is due to noise. Let's assume standard normal distribution ($\mu=0, \sigma=1$), then:

```
> m=numeric(1000)
> for ( i in 1:1000 ) {
  m[i]=mean(rnorm(20))
}
> m[1:8]
[1] -0.51381583  0.07614361  0.03961465 -0.22448761 -0.39037181  0.24697455 -0.10294914  0.14742867
> hist(m, col="pink")
> sum(abs(m)>0.3)
[1] 189
```

- In our simulation we have drawn 1000 random samples of size 20 from the normal distribution with mean 0 and variance 1 (i.e. under the null that assumes no effect). 189 of such samples (almost 20%) had mean that was greater than 0.3 by magnitude. Hence the observation made with the original sample is not significant



Distribution of the *sample means*
(remember, it's a random variable!)

TESTING FOR “DIFFERENCE FROM ZERO”

- One of the most ubiquitous tests (published in 1908 by William Gosset – a chemist at Guinness)
- Mechanics explained in previous slides (taking the shortcut of assuming known true variance)
- Analytical solutions do exist under certain assumptions (e.g. normality of the underlying distribution):
 - Student’s t-test; null hypothesis: the sample came from the (normal) distribution with zero mean. Small p-value means the null should be rejected and the mean is likely non-zero
 - Two-sample t-test: very similar, the null is that the two samples being compared came from the distribution(s) with the same mean. For instance, we can measure BMI in 50 males and 50 females and compare the two samples using the t-test. Small p-value->means are likely different
 - In R: `t.test()`

```
> t.test(rnorm(20))
```

```
One Sample t-test
```

```
data:  rnorm(20)
t = -1.0562, df = 19, p-value = 0.3041
alternative hypothesis: true mean is not equal to 0
95 percent confidence interval:
 -0.7551687  0.2486130
sample estimates:
mean of x
-0.2532779
```

SUMMARY

- In the first introductory lecture we looked at R datatypes, vectors, matrices and, importantly, powerful indexing capabilities. Review those and try playing with them on your own
- We reviewed basic statistical concepts:
 - Random variables
 - Probability distributions (discrete) and probability distribution functions (continuous)
 - Expected values, population mean and variance (those derived from the distribution itself, i.e. the *precise* values)
 - Estimators: estimating unknown population characteristics (mean and variance in our examples) *from a finite random sample*
 - Sampling error: sample mean and sample variance (and any other statistic we compute *from a sample*) are random variables themselves and are subject to (random!) change from one sample to the next
 - Hypothesis testing: given that we only have a (random) sample and its mean (or any other statistic) is subject to random sampling error, how can we tell that observed effect or trend (e.g. mean being non-zero) is a real effect rather than just the result of a random fluctuation? Hypothesis testing framework is a classical way of answering this question.
 - Joint, marginal, and conditional probabilities; statistical independence

ADDITIONAL DETAILS

- Above covers bare minimum of theory and practicalities of using R / R Studio necessary to complete this week assignment
- The following slides touch on adjacent topics that you will find useful later in the course

JOINT, MARGINAL AND CONDITIONAL PROBABILITIES

- Imagine the following scenario: Alice brought a bag of fruit from the grocery store. The bag contains apples and pears, some are green and some are red (the counts are given in the table below on the left). Bob reaches into the bag without looking: what are the probabilities of different outcomes?

- Here each “event” includes two observations: type of fruit (F) and color (C)

Probabilities:

Counts	F=Apple	F=Pear	Total
C=green	5	7	12
C=red	6	2	8
Total	11	9	20

	F=Apple	F=Pear	Marginal
C=green	0.25	0.35	0.6
C=red	0.3	0.1	0.4
Marginal	0.55	0.45	1

- We can form different probabilities (consult with the tables above):
 - Joint probability distribution $P(C,F)$: simply describes the probability of each combined event in the “space” of the two variables. For instance, $P(C=\text{green}, F=\text{apple})=0.25$, $P(C=\text{red}, F=\text{apple})=0.3$, ... (body of the table)
 - Marginal probabilities $P(C)$, $P(F)$: of course we can still ask about the probability in the space of one variable completely ignoring the other, for instance $P(F=\text{apple})=0.55$, $P(C=\text{green})=0.6$, ... (in the *margins* of the table)
 - Conditional probabilities: the probabilities $P(C|F)$ of different colors when the fruit type is *given* or vice versa, the probabilities $P(F|C)$ of different fruits when we *know* the color; for instance $P(C=\text{green}|F=\text{apple})=5/11$, $P(F=\text{apple}|C=\text{red})=6/8$ (note, the denominators are the corresponding row/column *marginal* totals, not the grand total!).

Note how the knowledge that the fruit is red changes the probability (or our confidence in) it's an apple!

STATISTICAL INDEPENDENCE, BAYES THEOREM

- Two random variables X, Y are called *statistically independent* when $P(X, Y) = P(X)P(Y)$
- This can be better understood with the help of famous Bayes' theorem which states that:

$$P(X, Y) = P(X|Y)P(Y) = P(Y|X)P(X)$$

- For instance, in our earlier example $P(C=\text{green}|F=\text{apple})=5/11$, while $P(F=\text{apple})=11/20$, thus $P(C=\text{green}, F=\text{apple})=5/11 * 11/20 = 5/20 = 0.25$, which is exactly what our table shows.
- Combining the definition of statistical independence with Bayes theorem we immediately see that

$$P(X)P(Y) = P(X|Y)P(Y) = P(Y|X)P(X), \text{ thus}$$

$$P(Y|X) = P(Y) \text{ and } P(X|Y) = P(X)$$

In other words variables are statistically independent when **conditional probability distribution equals the marginal probability distribution** (in other words it in fact does not depend on the “condition” placed on the other variable, hence “independence”)

ASSOCIATION TESTS ON CONTINGENCY TABLES

- In our toy model **IF** we know exact probabilities (as shown in the table), then variables C(olor) and F(ruit) are *not* independent
 - What is the scenario, in which the probabilities as shown *are* the ultimate truth?
 - If that bag of fruit is a *sample* and we use it to learn something about the bigger population, what then? (hint: sampling error)
 - Tests for association that can be applied to contingency tables: chi-square, Fisher exact test
 - Tests faithfully measure significance, while experimental design is our problem: in the described scenario, what is the population the bag of fruit represents? What are we learning about? Something about fruit grown in the US? Or about how supermarket stocks its grocery department based on its own marketing research? Or our sample is biased by Alice's own preferences?

```
> chisq.test(matrix(c(5,6,7,2),ncol=2))
```

```
      Pearson's Chi-squared test with Yates' continuity correction
```

```
data:  matrix(c(5, 6, 7, 2), ncol = 2)
X-squared = 1.0185, df = 1, p-value = 0.3129
```

```
> fisher.test(matrix(c(5,6,7,2),ncol=2))
```

```
      Fisher's Exact Test for Count Data
```

```
data:  matrix(c(5, 6, 7, 2), ncol = 2)
p-value = 0.1968
alternative hypothesis: true odds ratio is not equal to 1
...
```

What if the sample showed exactly the same proportions but was just bigger? Exercise: multiply all counts by 10 and recompute p-value(s)

MODE EXAMPLES

- You can view the mode of any value, whether typed explicitly or stored in a variable, using the function `mode()`
- Note how we can assign values of different types to the same variable `x`: variables do not have types, their *values* do
 - Since variables do not have pre-defined, declared type, there is no guarantee regarding the type of the currently stored value: a great source of bugs (as in any dynamically typed language). When in doubt, **CHECK** what your variable holds!!

```
> mode(1)
[1] "numeric"
> mode("abc")
[1] "character"
> mode(TRUE)
[1] "logical"
> mode(7+5i)
[1] "complex"
```

```
> x<-5.5
> mode(x)
[1] "numeric"
# can re-assign to x a value of different type:
> x<-"text"
> mode(x)
[1] "character"
> y=list(2,TRUE,"abc") # this is how we create a list
> mode(y)
[1] "list"
```

SPECIAL VALUES

- R has a few special values. All of them are “true values” in a sense that they are not just printed into output when something bad happens, but they can be assigned to a variable, passed to a function as arguments, compared, etc.
 - Inf, -Inf: positive/negative infinity. Result is too large (by magnitude) and cannot be represented
 - NaN: not a number, result makes no sense (e.g. subtracting two numbers that are both too large, what’s the result??)
 - NULL: a special value that represents “nothing”, an “empty value” (examples will follow)
 - NA: “not available”, arguably the most important special value in R; you will encounter it often, you will love it and you will hate it. Represents “missing value” in experimental data (i.e. we tried to measure something but measurement failed, the notebook was lost, disk burnt, etc). **NOT** the same as NULL and behaves very differently! (examples will follow)

```
> 3^1000
[1] Inf
> -3^1000
[1] -Inf
> 1/0
[1] Inf
> 2 < Inf
[1] TRUE
```

```
> 2 < -Inf
[1] FALSE
# we can type Inf, it is a
# legitimate value literal:
> Inf - Inf
[1] NaN
> 0/0
[1] NaN
```

```
> NA==1 # can you explain results?
[1] NA
> 1+NA
[1] NA
> NA | TRUE
[1] TRUE
> NA & TRUE
[1] NA
```


TESTING FOR SPECIAL VALUES

- Always use dedicated functions to test for special values
 - All special values are legitimate values in the language in a sense that they can be returned, stored in a variable, etc. You can test whether a variable contains a special value in the same way we are testing the literals below

```
# here we ask if some unknown (missing) value is equal to another missing value.
# The answer is "I do not know", naturally!
> NA == NA
[1] NA
> is.na(NA) # this is how we ask if the value is a missing one!!
[1] TRUE
> is.null(NULL) # check if the value is NULL
[1] TRUE
> is.nan(NaN) # check if the value is NaN
[1] TRUE
> is.infinite(Inf) # check if the value is Inf
[1] TRUE
> x=NA; is.na(x) # testing if variable contains the special value
[1] TRUE
```

TYPE CONVERSION

- Conversion from one mode to another is known in R as “coercion”
- Many languages have array-like objects (often called lists) that can (at least potentially) hold values of different types
 - R’s **lists** are like that but not vectors! In R, a vector of any specific mode holds only values of that mode
 - Attempt to append/insert a value of more general mode will change the mode of the vector
 - ... or one can change the mode manually and explicitly

```
> x=1:5
# never mind that character string "6" could be converted to a number. We add a
# character value, the vector changes its mode to 'character' without much thinking:
> c(x, "6")
[1] "1" "2" "3" "4" "5" "6"
# we can enforce conversion to integer:
> as.integer(c(x, "6")) # as.numeric() also exists
[1] 1 2 3 4 5 6
> as.character(1:3)
[1] "1" "2" "3"
```

R indicates that values are of character type by using quotation marks in the output. You can use `mode()` to confirm that the vector is indeed of character type

TYPE CONVERSION (CONTINUED)

- What happens if one appends/inserts a value of *less* general mode into a vector?
 - The value will become converted to the more general mode (current mode of the vector), the mode of the vector will not change

```
> x=as.character(1:5); x # create a vector of character values
[1] "1" "2" "3" "4" "5"
> c(x,6.2) # try concatenating with a number (or rather with a numeric vector of length 1)
[1] "1" "2" "3" "4" "5" "6.2"
> c(x,TRUE,F) # concatenate with logical values - the latter become promoted to character
[1] "1" "2" "3" "4" "5" "TRUE" "FALSE"
# when logical values are concatenated onto a numeric vector, the former become
# numbers (TRUE is 1 and FALSE is 0):
> c(seq(2,8,by=2),T,F)
[1] 2 4 6 8 1 0
```

INDEXING A VECTOR (“BY NAME”)

- Elements of a vector in R (and of other objects as we will see later) can be *named*. One can assign names manually; alternatively, some functions return their results as vectors (or other objects) with named elements

```
> y <- 2:6; y # create a vector (and print it); this part we know already
[1] 2 3 4 5 6
> names(y) <- LETTERS[1:5]; y # we can assign names to elements! Print and see
A B C D E
2 3 4 5 6
> y[c("B", "D", "B")] # now we can also access elements by name
B D B
3 5 3
```

OUT OF BOUND INDEXING

- When index (of any kind: positional, logical or by name) is out of bounds, *no error is generated*. Instead, NA is returned for missing elements. When index is NA, the selected element is also NA:

```
> clr[c(2,6,3)] # select subset with one index out of bounds
[1] "blue" NA "gray"
> clr[c(2,NA,3)] # select a subset with one index unknown
[1] "blue" NA "gray"
> y=2:6; names(y)=LETTERS[1:5]; y[c("B","D","F")]
B    D  <NA>
3    5   NA
> clr[c(T,F,F,T,F,T)] # indexing vector too long, so the last T out of bounds: returns NA
[1] "red" "pink" NA
> clr[c(T,F,F,T,F,F)] # compare: indexing vector too long, but the out of bounds element is F
[1] "red" "pink"
# NA is logical!! Here it's unknown whether to take the 5th element or not, so we get NA:
> clr[c(T,F,F,T,NA)]
[1] "red" "pink" NA
```

OUT OF BOUND INDEXING IN ASSIGNMENTS

- When indexing expression appears on the left hand side (lhs) of an assignment and out of bound indexes are present, the target vector of the assignment is automatically and silently *expanded* to include the requested index; if there is space between current last element and out of bound element being added, this space is filled with NA

```
> z=1:3
# assign to elements at positions 5, 6. Note that element at position 4 remains uninitialized, i.e.
# gets the "missing value" (NA)
> z[5:6]=c(-1,-2); z
[1] 1 2 3 NA -1 -2
# here we select the 4th element (currently NA) for the assignment, as well as the 8th and the 10th -
# the latter are out of bounds! Vector z is promptly expanded so now it has length 10, and the 7th
# and the 9th elements that we chose to skip get auto-initialized with NA:
> z[c(F,F,F,T,F,F,F,T,F,T)]=0; z
[1] 1 2 3 0 -1 -2 NA 0 NA 0
```

USEFUL FUNCTIONS (CONTINUED)

■ More functions:

```
# are 2 and 10 present among elements of xx? Returned vector
# has the same length as the 1st arg and for each element x of
# that arg it tells if x is among the values in the 2nd arg:
> is.element(c(2,10),xx)
[1] TRUE FALSE
# at what positions do 0, -2, 10 occur in xx? Returned vector
# has the same length as the 1st arg:
> match(c(0,-2,10),xx)
[1] 3 1 NA
# same as is.element(c(0,-2,10),xx) (can you see why?) :
> ! is.na(match(c(0,-2,10),xx))
[1] TRUE TRUE FALSE
> any(xx > 0) # is at least one element of xx greater than 0?
[1] TRUE
> all(xx > 0) # are all elements of xx greater than 0?
[1] FALSE
> any(xx > 2) # is at least one element of xx greater than 2?
[1] NA
```

Why? Remember, xx is
(-2 -1 0 1 2 NA)

```
> max(xx) # min() also exists of course
[1] NA
> max(xx,na.rm=1)
[1] 2
# what is the index of the largest elem?:
> which.max(xx) # inconsistent use of NA!
[1] 5
# create another vector:
> yy <- c(3,-3,5,-5,4,4)
> sort(yy) # sort a vector
[1] -5 -3 3 4 4 5
# find sorting order for the argument:
> order(yy)
[1] 4 2 1 5 6 3
# reshuffling a vector into its sorting
# order IS sorting!:
> yy[order(yy)]
[1] -5 -3 3 4 4 5
```