

# Rbootcamp/Workshop

Michael Kleinsasser

8/27/2019

# Mike's Personal Introduction

- R programmer for the Department of Biostatistics
- Write and maintain R packages for faculty and students
- Consult faculty and students on writing R packages, optimization



# Rbootcamp Introduction

Goals (roughly):

- Day 1: Setting up and using R interactively via Rstudio. Basic syntax and data types. Creating, viewing, and removing objects. The on-line help. Writing R functions. How most statistical procedures are implemented in R
- Day 2: Using R non-interactively on a cluster. How to write optimized code, including vectorization and loops. Introduction to data manipulation with dplyr and visualization with ggplot2
- Day 3: Finally, we combine our knowledge and apply it to a series of simulation problems

# Materials

- All bootcamp materials online at <https://github.com/umich-biostatistics/Rbootcamp>
  - Handouts with examples to work through
  - R scripts of our examples (.Rmd slides, .R scripts)
- Go to link and download zip archive, extract

# Setup

Go to Rstudio cloud to follow along:

- Enter username, etc. for free account
- Follow along by typing commands in my slides
- If you have Rstudio/R, open the .Rmd to follow along
- Recommended: Install R/Rstudio asap
- See course materials for download instructions

# R Basics: Big Picture

- R is a sophisticated calculator for statistics

Chambers (2016) Extending R:

- Everything that exists in R is an object
- Everything that happens in R is a function call

Obtain a basic working knowledge of R objects and functions,

- Google the rest!

# R Basics: a basic schematic view

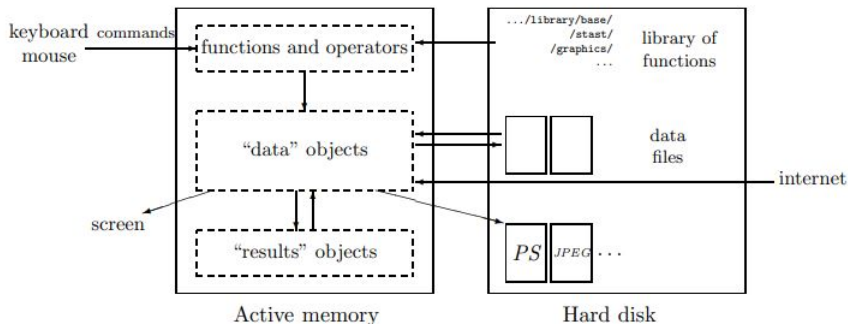


Figure 1: A schematic view of how R works.

# R Basics: Goals

- Data types and functions
  - Create object having data types
  - Combine those into data structures
  - Write basic R function
- Learn parts of R most useful to statisticians
  - How do most modeling functions work in R, and
  - How to inspect structure and content of objects
  - Data manipulation and visualization
- Apply knowledge to simulation
  - Set up, draw from various distributions
  - Summarise, visualize



# Ways to run R

Either “interactively” or “non-interactively”

Interactive R:

- For line-by-line code execution
- Useful for data exploration, debugging, etc.
- Open the Rgui.exe (on Windows) or
- Open Rstudio (recommended) - integrated development environment for R

Non-interactively:

- For running entire scripts at once
- Desired for allocating large jobs on the cluster (return to later)

## Use R as a calculator

Standard operations:

```
# multiply *, divide /, add + subtract -
18056.983 - 1005.118 + 22.53
```

```
## [1] 17074.4
```

$$\left( \frac{\pi - 3.14}{3.14} \right) * 100$$

```
## [1] 0.05072145
```

- Can do anything your Ti84 does, and then some



# Operators: arithmetic and logical

Operator	Description
+	addition
-	subtraction
/	division
^	exponential
%%	modulus (x mod y)
%/%	integer division

**Table 1:** Arithmetic Operators

Operator	Description
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal
==	exactly equal to
!=	not equal to
!x	not x (x logical)
x y	x OR y
x&y	x AND y
isTRUE(x)	is x TRUE

**Table 2:** Relational/Logical Operators

# Operators: an example

Test if these two expressions are equivalent in R:

- use Ctrl + Enter to execute each desired line

```
A = c(TRUE, TRUE, FALSE); B = c(FALSE, FALSE, TRUE)
```

```
# Expression 1:
```

```
A | B
```

```
# Expression 2:
```

```
!(A == B)
```

```
# test equality:
```

```
all.equal(A | B, !(A == B))
```

# Operators and vectorization

Arithmetic/logical operators are **vectorized**: vector in -> vector out

Given these vectors:

```
## [1] 1 2 3
```

```
## [1] 4 8 16
```

```
x + y
```

```
## [1] 5 10 19
```

```
x * y
```

```
## [1] 4 16 48
```

```
y <= x
```

```
## [1] FALSE FALSE FALSE
```

# Create, list, delete objects

Create object with “assign” operator

- arrow then minus sign <-
- single equal sign =

```
# x gets the number 3.14
```

```
x <- 3.14
```

```
x      # print x
```

```
## [1] 3.14
```

```
# equivalently
```

```
x = 3.14
```

```
x      # print x
```

```
## [1] 3.14
```

# Create, list, delete objects

Objects we create are stored in memory, e.g.:

```
name = "Carmen"  
n1 = 10  
n2 = 100  
m = 0.5
```

Use `ls()` function to list all objects in memory:

```
ls()
```

```
## [1] "m"      "n1"     "n2"     "name"   "x"      "y"
```

Notice: I created `x` before, it's still in memory.

# Create, list, delete objects

Use `ls()` function to list all objects in memory:

```
ls()
```

```
## [1] "m"      "n1"     "n2"     "name"   "x"      "y"
```

The function `ls.str()` displays some details about objects in memory:

```
ls.str()
```

```
## m :   num 0.5
## n1 :   num 10
## n2 :   num 100
## name :  chr "Carmen"
## x :   num 3.14
## y :   num [1:3] 4 8 16
```



# Create, list, delete objects

To delete objects in memory, use `rm()` function

```
rm(x) # deletes object named x  
ls()  # which objects remain in memory?
```

```
## [1] "m"      "n1"     "n2"     "name"  "y"
```

```
rm(m, n2, name) # remove multiple objects  
ls()
```

```
## [1] "n1" "y"
```

```
rm(list = ls()) # remove everything from memory  
ls()
```

```
## character(0)
```

# The on-line help

R has structured help pages providing “how-to”

- **Description:** what function does
- **Usage:** name with arguments and options
- **Arguments:** how each argument should be structured
- **Details:** more detailed description
- **Value:** How the output is structured/ what it contains
- **Examples:** examples of the function in use

```
?rm           # help documentation for rm function  
help("rm")    # alternately
```

`remove {base}`

R Documentation

## Remove Objects from a Specified Environment

### Description

`remove` and `rm` can be used to remove objects. These can be specified successively as character strings, or in the character vector `list`, or through a combination of both. All objects thus specified will be removed.

If `envir` is `NULL` then the currently active environment is searched first.

If `inherits` is `TRUE` then parents of the supplied directory are searched until a variable with the given name is encountered. A warning is printed for each variable that is not found.

### Usage

```
remove(..., list = character(), pos = -1,  
        envir = as.environment(pos), inherits = FALSE)  
  
rm      (... , list = character(), pos = -1,  
         envir = as.environment(pos), inherits = FALSE)
```

## Other R help

Many package writers create Vignettes and READMEs

How to view vignettes?

```
vignette(all = TRUE) # list vignettes for installed packages  
vignette(all = FALSE) # vignettes from attached packages
```

How to view READMEs?

Other help...

# Objects in R

Objects can be data, model output, functions

- Characterized by their **names** and **content**
- attributes - specify the kind of data represented
  - e.g. mode, length

```
x = 25
```

```
mode(x)
```

```
## [1] "numeric"
```

```
length(x)
```

```
## [1] 1
```

Example: What are the mode and length of this object?

```
l = c(3-2i, 5+2i, 8-2i)
```

# Basic data “modes” in R

The mode is the basic type of the elements of an object

The four main modes:

- numeric, comes in two flavors: integer, numeric
- character
- logical
- complex

```
num = 15.533; name = "Mike"; isStudent = TRUE
```

```
mode(num); mode(name); mode(isStudent)
```

```
## [1] "numeric"
```

```
## [1] "character"
```

```
## [1] "logical"
```

# Atomic vectors

Fundamental data structure in R:

- atomic vector - vector in which every element is of same mode

To create an atomic vector, use `c()` function:

```
c(3.145, 2.18, 9.98e3, 0.05)
```

```
## [1]      3.145      2.180 9980.000      0.050
```

Example: Store the above numeric vector by assigning it a name

Example: create empty character vector of length 3 and store your full name

```
vector(mode = "character", length = 3)
```

```
## [1] "" "" ""
```

# Data types examples

3 ways to create numeric vector:

```
# empty numeric vector  
y1 <- numeric(6)  
y1      # print y1
```

```
## [1] 0 0 0 0 0 0
```

```
y2 <- vector(mode = "numeric", length = 6)  
y2      # print y2
```

```
## [1] 0 0 0 0 0 0
```

```
y3 <- c(5, 13.222, 2, 0.001, 77.4, 31.9)  
y3      # print y3
```

```
## [1] 5.000 13.222 2.000 0.001 77.400 31.900
```



# Objects in R: NA

NA means “Not Available” and it denotes missing data

```
c(3, 5, 9, NA, 18, 25, NA)
```

```
## [1] 3 5 9 NA 18 25 NA
```

Example: Store the vector above and remove the missing values.

```
#incomplete.data =  
#complete.data = incomplete.data[complete.cases(incomplete.data)]
```

# R data types/structures

Four fundamental data types:

- character, numeric (numeric or integer), logical, complex

Combine to form data structures

- atomic vector (atomic - vector of single type)
- list
- matrix
- data.frame
- factor

We will focus on matrices and data.frames

# Matrices

Matrices are the natural extension of atomic vectors into 2 dimensions

- any mode can be used, but numeric most common:

Syntax:

```
m = matrix(data, nrow, ncol, byrow, dimnames)
```

- **data** is the input vector which becomes the data elements of the matrix
- **nrow, ncol** is the number of rows/columns to be created
- **byrow** is T/F. If TRUE then the input vector elements are arranged by row
- **dimnames** is the names assigned to the rows and columns

# Matrix examples:

Identity matrix:

```
dat = c(1,0,0,0,1,0,0,0,1)  # data
iden = matrix(data = dat, nrow = 3, byrow = T)
iden  # print matrix
```

```
##      [,1] [,2] [,3]
## [1,]    1    0    0
## [2,]    0    1    0
## [3,]    0    0    1
```

Easier:

```
iden = diag(rep(1,3))
iden
```

## access elements of a matrix

- single brackets used to access elements

Access individual elements:

```
# 2x5 matrix of numbers 1 to 10  
P = matrix(data = 1:10, nrow = 2)  
P[1,3] # row 1, column 3
```

```
## [1] 5
```

```
P[nrow(P),ncol(P)] # row 2, column 5 (bottom right position)
```

```
## [1] 10
```

Access entire rows/columns:

```
P[,3];
```

```
## [1] 5 6
```

# The data.frame

The **data.frame** is the most common way to store and work with data in R

- Not surprising: they are designed for this purpose
- Most modeling functions work on data.frames

Composed of a list of equal length atomic vectors (can be of any type)

Exercise:

The following are data on students in the class:

- Has Master's (logical): TRUE FALSE FALSE TRUE
- GPA (numeric): 3.1 4.0 2.9 3.6
- First Name (character): Mike Dan Sara Karen

Convert to three atomic vectors of appropriate type.

```
#insert solution
```

# data.frame examples

Create a data.frame out of the following “class” data:

- Has Master's (logical): TRUE FALSE FALSE TRUE
- GPA (numeric): 3.1 4.0 2.9 3.6
- First Name (character): Mike Dan Sara Karen

```
# store data
has_ms <- c(TRUE, FALSE, FALSE, TRUE)
gpa <- c(3.1, 4.0, 2.9, 3.6)
name <- c("Mike", "Dan", "Sara", "Karen")
# Create data.frame
dat <- data.frame(has_MS = has_ms, GPA = gpa, Name = name)
dat      # print data.frame
```

```
##   has_MS GPA  Name
## 1   TRUE 3.1  Mike
## 2  FALSE 4.0   Dan
## 3  FALSE 2.9   Sara
## 4   TRUE 3.6 Karen
```

## access elements of a data.frame

Each vector of a data.frame contains the values of a variable

Access each vector with the dollar sign \$

Ex: Extract the GPA column and print it

```
dat$GPA
```

```
## [1] 3.1 4.0 2.9 3.6
```

Another example: ToothGrowth data.

```
ToothGrowth$len
```

Exercise: Add a new column to the data.frame dat with NAs

*# Hint: = NA will recycle NA to the appropriate length*



## Preview data.frame

Preview head (first few rows) of data.frame:

```
head(ToothGrowth)
```

```
##      len supp dose
## 1  4.2   VC  0.5
## 2 11.5   VC  0.5
## 3  7.3   VC  0.5
## 4  5.8   VC  0.5
## 5  6.4   VC  0.5
## 6 10.0   VC  0.5
```

View tail of data.frame:

```
tail(dat)
```

View entire data.frame in new window:

```
View(ToothGrowth)
```

# Inspect an object

- `class()` - what kind of object is it (high-level)?
- `typeof()` - what is the data type (low-level)?
- `length()` - how long is it?
- `attributes()` - does it have meta-data?

# Inspect an object

- `class()` - what kind of object is it (high-level)?

```
class(ToothGrowth)
```

```
## [1] "data.frame"
```

- `typeof()` - what is the data type (low-level)?

```
typeof(ToothGrowth$supp)
```

```
## [1] "integer"
```

# Inspect an object

- `length()` - how long is it?

```
length(ToothGrowth$dose)
```

```
## [1] 60
```

- `attributes()` - does it have meta-data?

```
attributes(ToothGrowth)
```

```
## $names
```

```
## [1] "len" "supp" "dose"
```

```
##
```

```
## $class
```

```
## [1] "data.frame"
```

```
##
```

```
## $row.names
```

```
## [1] 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
```

# access elements of different data types

Access elements of any...

❶ vector: [index]

```
dose1 = ToothGrowth$dose  
does1[3]
```

❷ matrix: [row index, col index]

```
P[2,2]
```

❸ list: [[index]]

```
ToothGrowth[[1]]  
# data.frames are technically lists  
# can you identify what this is?
```

❹ data.frame: \$name

# Summary of data types

- Create objects having different data types
- Combine those into data structures
- access elements of interest

# R functions

R function syntax:

```
NAME <- function(ARG1, ARG2, ARG3) {  
  DO SOMETHING  
  STORE RESULT  
  return(RESULT)  
}
```

```
pow <- function(base, expon) { # power function  
  prod(rep(base, expon)) # base^(expon)  
}  
# Use power function  
pow(5, 2)
```

```
## [1] 25
```

```
pow(10, 3)
```

```
## [1] 1000
```

# R functions:

R function syntax:

```
NAME <- function(ARG1, ARG2, ARG3) {  
  DO SOMETHING  
  STORE RESULT  
  return(RESULT)  
}
```

Exercise: Write a function that calls the `pow()` function and returns a list of base taken to the powers 2, 4, and 8.

*# Hint: create list with `list(pow2 = , pow4 = , pow8 = )`*



## R functions: exercises

Write a function that takes no arguments and adds 1 to 'a' if it exists, or sets a = 1 if it does not exist. `exists()` returns TRUE if its argument exists in memory and zero else.

```
# Here is a template to start with
add1 = function() {
  if(# test if a does NOT exist) {
    # set a equal to 1
  } else {
    # add one to a
  }

  return(#return the correct object)
}
```

## R functions: exercises

Does your function work? Test it with the following scenarios.

```
# Run add1(), should return 1
```

```
# Set a = 5, add1() should return 6
```

```
# Set a = -1, add1() should return 0
```

# Common R functions

R has a huge collection of packages:

- 6,000+ packages for data analysis build (on CRAN alone)

Example: `lm` (linear models)

- Use `?lm` to read help documentation

`lm {stats}`

R Documentation

## Fitting Linear Models

### Description

`lm` is used to fit linear models. It can be used to carry out regression, single stratum analysis of variance and analysis of covariance (although [aov](#) may provide a more convenient interface for these).

### Usage

```
lm(formula, data, subset, weights, na.action,  
   method = "qr", model = TRUE, x = FALSE, y = FALSE, qr = TRUE,  
   singular.ok = TRUE, contrasts = NULL, offset, ...)
```

### Arguments

- |                      |  |
|----------------------|--|
| <code>formula</code> | an object of class " <a href="#">formula</a> " (or one that can be coerced to that class): a symbolic description of the model to be fitted. The details of model specification are given under 'Details'.   |
| <code>data</code>    | an optional data frame, list or environment (or object coercible by <a href="#">as.data.frame</a> to a data frame) containing the variables in the model. If not found in <code>data</code> , the variables are taken from <code>environment(formula)</code> , typically the environment from which <code>lm</code> is called. |

# Fit a linear model with lm

- Use built-in data set ToothGrowth
- ?ToothGrowth for help:

## The Effect of Vitamin C on Tooth Growth in Guinea Pigs

### Description

The response is the length of odontoblasts (cells responsible for tooth growth) in 60 guinea pigs. Each animal received one of three dose levels of vitamin C (0.5, 1, and 2 mg/day) by one of two delivery methods, orange juice or ascorbic acid (a form of vitamin C and coded as VC).

### Usage

```
ToothGrowth
```

### Format

A data frame with 60 observations on 3 variables.

```
[,1] len    numeric Tooth length  
[,2] supp   factor   Supplement type (VC or OJ).  
[,3] dose   numeric Dose in milligrams/day
```

# View the data

View data in new window:

```
View(ToothGrowth)
```

Or use head to view only first 6 rows:

```
head(ToothGrowth)
```

```
##      len supp dose
## 1  4.2   VC  0.5
## 2 11.5   VC  0.5
## 3  7.3   VC  0.5
## 4  5.8   VC  0.5
## 5  6.4   VC  0.5
## 6 10.0   VC  0.5
```

How big is the data?

```
dim(ToothGrowth)
```

```
## [1] 60  3
```

# Using lm() function for linear models

Call lm on the data and formula, store result "lm" object:

```
tooth_fit = lm(formula = len ~ supp + dose,  
               data = ToothGrowth)
```

Formulas in R:

```
len ~           # Response column name, ~ for "="  
  supp +       # First predictor name + for "+"  
  dose         # second predictor name
```

Many R functions use the formula argument.

# Getting detailed information

Basic “print” of model:

```
print(tooth_fit)      # equivalent to tooth_fit

##
## Call:
## lm(formula = len ~ supp + dose, data = ToothGrowth)
##
## Coefficients:
## (Intercept)      suppVC          dose
##      9.272      -3.700       9.764
```

Detailed summary:

```
summary(tooth_fit)

##
## Call:
## lm(formula = len ~ supp + dose, data = ToothGrowth)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -6.600 -3.700  0.373  2.116  8.800
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   9.2725     1.2824   7.231 1.31e-09 ***
## suppVC       -3.7000     1.0936  -3.383  0.0013 **
## dose          9.7636     0.8768  11.135 6.31e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

# Understanding R classes

- What is this thing?

```
class(tooth_fit)
```

- What are the methods for this object?

```
methods(class = "lm")
```

- What is its structure? (i.e., what's in it)

```
str(tooth_fit)
```



# Understanding R classes

- What is this thing?

```
class(tooth_fit)
```

```
## [1] "lm"
```

# Understanding R classes

- What are the methods for this object?

```
methods(class = "lm")
```

```
## [1] add1          alias          anova          case.names
## [5] coerce        confint        cooks.distance deviance
## [9] dfbeta        dfbetas       drop1          dummy.coef
## [13] effects       extractAIC    family        formula
## [17] hatvalues     influence     initialize     kappa
## [21] labels        logLik        model.frame    model.matrix
## [25] nobs          plot          predict        print
## [29] proj          qr            residuals      rstandard
## [33] rstudent      show          simulate       slotsFromS3
## [37] summary       variable.names vcov
## see '?methods' for accessing help and source code
```

# Understanding R classes

- What is its structure? (i.e., what's in it)

```
str(tooth_fit)
```

```
## List of 13
## $ coefficients : Named num [1:3] 9.27 -3.7 9.76
## ..- attr(*, "names")= chr [1:3] "(Intercept)" "suppVC" "dose"
## $ residuals : Named num [1:60] -6.25 1.05 -3.15 -4.65 -4.05 ...
## ..- attr(*, "names")= chr [1:60] "1" "2" "3" "4" ...
## $ effects : Named num [1:60] -145.73 14.33 47.16 -3.86 -3.26 ...
## ..- attr(*, "names")= chr [1:60] "(Intercept)" "suppVC" "dose" "" ...
## $ rank : int 3
## $ fitted.values: Named num [1:60] 10.5 10.5 10.5 10.5 10.5 ...
## ..- attr(*, "names")= chr [1:60] "1" "2" "3" "4" ...
## $ assign : int [1:3] 0 1 2
## $ qr :List of 5
## ..$ qr : num [1:60, 1:3] -7.746 0.129 0.129 0.129 0.129 ...
## .. ..- attr(*, "dimnames")=List of 2
## .. .. ..$ : chr [1:60] "1" "2" "3" "4" ...
## .. .. ..$ : chr [1:3] "(Intercept)" "suppVC" "dose"
## .. ..- attr(*, "assign")= int [1:3] 0 1 2
## .. ..- attr(*, "contrasts")=List of 1
## .. .. ..$ supp: chr "contr.treatment"
## ..$ qraux: num [1:3] 1.13 1.11 1.11
## ..$ pivot: int [1:3] 1 2 3
## ..$ tol : num 1e-07
## ..$ rank : int 3
## ..- attr(*, "class")= chr "qr"
## $ df.residual : int 57
## $ contrasts :List of 1
## ..$ supp: chr "contr.treatment"
## $ xlevels :List of 1
```

# Understanding R classes

- Pull something out of the “lm” fit object:

```
tooth_fit$fitted.values      # y_hat's for the linear model
```

```
##      1      2      3      4      5      6      7      8
## 10.45429 10.45429 10.45429 10.45429 10.45429 10.45429 10.45429 10.45429
##      9     10     11     12     13     14     15     16
## 10.45429 10.45429 15.33607 15.33607 15.33607 15.33607 15.33607 15.33607
##     17     18     19     20     21     22     23     24
## 15.33607 15.33607 15.33607 15.33607 25.09964 25.09964 25.09964 25.09964
##     25     26     27     28     29     30     31     32
## 25.09964 25.09964 25.09964 25.09964 25.09964 25.09964 14.15429 14.15429
##     33     34     35     36     37     38     39     40
## 14.15429 14.15429 14.15429 14.15429 14.15429 14.15429 14.15429 14.15429
##     41     42     43     44     45     46     47     48
## 19.03607 19.03607 19.03607 19.03607 19.03607 19.03607 19.03607 19.03607
##     49     50     51     52     53     54     55     56
## 19.03607 19.03607 28.79964 28.79964 28.79964 28.79964 28.79964 28.79964
##     57     58     59     60
## 28.79964 28.79964 28.79964 28.79964
```

# Extracting data from model objects

Some generic extraction methods:

```
coef(tooth_fit)           # model coefficients

coef(summary(tooth_fit))  # adds test statistics, p-values

vcov(tooth_fit)           # variance/covariance matrix
```

Note: depending on implementation, these may not be available - Check methods with “methods(object)” before attempting

# Extracting data from model objects (in detail)

Extract coefficients, test stats, and p-values

```
coef(summary(tooth_fit))    # adds test statistics, p-values
```

	Estimate	Std. Error	t value	Pr(> t )
## (Intercept)	9.272500	1.2823649	7.230781	1.312335e-09
## suppVC	-3.700000	1.0936045	-3.383307	1.300662e-03
## dose	9.763571	0.8768343	11.135025	6.313519e-16

# Predict new values

- `predict()` function is generic and works with many models
- Pass in a new `data.frame` with the same column names:

```
to_predict = data.frame(dose = 0.5, supp = "VC")
```

```
predict(tooth_fit, newdata = to_predict)
```

```
##           1
```

```
## 10.45429
```

## Predict new values (example 2)

- `predict()` function is generic and works with many models
- Pass in a new `data.frame` with the same column names:

```
to_predict = data.frame(dose = seq(0,1,0.1), supp = "OJ")
```

```
predict(tooth_fit, newdata = to_predict)
```

```
##           1           2           3           4           5           6           7
##  9.27250 10.24886 11.22521 12.20157 13.17793 14.15429 15.13064 16.10708
##           9          10          11
## 17.08336 18.05971 19.03607
```



## Day 2

- Non-interactive R and the cluster (Instructed by Dan Barker)
- How to write optimized code, including vectorization and loops, and
- introduction to data manipulation with dplyr, and visualization with ggplot2

# Packages to install for today

Run the following chunk to install packages:

```
install.packages("tidyverse")
```

Load into memory:

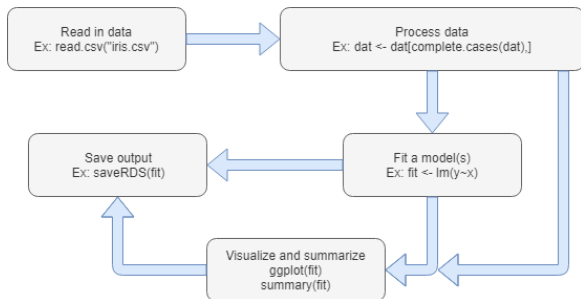
```
library(tidyverse) # actually a set of packages
```

# R on cluster (non-interactive R)

## Cluster Computation

- Dan Barker danbarke at umich.edu
- Cluster System Administrator

# Basic workflow for statistical analysis



# What is the tidyverse?

- collection of (very useful) R packages for doing “data science”
- ggplot2 - for plotting, visualizing data
- dplyr - for data manipulation/processing
- tidyr, readr (read data), purrr (better loops), tibble (improved data.frame), stringr, forcats
- Learn: R for Data Science, Grolemund and Wickham (free online)

# Advanced control structures

Common misconception: “loops in R are slow”

- Many commenters say loops in R are a bad idea,
- But, sometimes difficult (or impossible) to write vectorized code, or vectorized code consumes too much memory.
- We discuss:
  - How to improve loops when they are necessary
  - Eliminate them when possible

## Loops: often not necessary

- Many R functions are “vectorized” (vector in, vector out)
- While most other languages require loops, R does not:

```
a = c(5, 2, 4, 12, 1)
```

```
b = c(2, 0, 3, -1, 2)
```

```
a + b      # vector + vector = vector
```

```
## [1]  7  2  7 11  3
```

Takehome: When possible, operate on vectors and matrices, don't loop over each row/position index

## Problem: not all functions are vectorized

- Some functions, like `read.table()` for reading a table of data into R, are not vectorized
- But what if we have a list of files to read in? “data1.txt”, “data2.txt”, “data3.txt”, ..., “data50.txt”

```
# the 50 data set names  
file_names = paste0("data", 1:50, ".txt")
```

Attempt it, will cause error:

```
read.table(file_names) # error!
```



## map() from purrr package to avoid loop

- map() allows you to apply a function to each element of a vector
- faster, easier to read than a loop

```
read.table(file_names)  # error!
```

```
# list of 50 data sets  
my_dat_list = map(file_names, read.table)  
# results from reading data1.txt  
my_dat_list[[1]]  
# results from reading data50.txt  
my_dat_list[[50]]
```

# Create your own functions for map

Generate random samples from a Normal with different variances

```
draws = map(2:20, function(x) rnorm(25, mean=0, sd=x))  
# returns a list with vectors of draws  
str(draws)  
# first vector of draws:  
draws[[1]]  
  
# last vector of draws:  
draws[[19]]
```

## Create your own functions for map

Now, estimate the standard error:

```
map(draws, sd)  # sd() is standard deviation in R
```

```
map_dbl(draws, sd) # numeric vector
```

## For loop vs map() version

- Standard for loop:

```
sdvs = 2:20; result.list = list(length = 19)
for (i in 1:19) {
  result.list[[i]] = rnorm(25, mean=0, sd=sdvs[i])
}
```

- map():

```
map(2:20, function(x) rnorm(25, mean=0, sd=x))
```

Advice: If you're tracking indexes (like with the for loop), consider re-writing so you no longer have to depend on correct indexing

# Sometimes loops are required

Sometime you just need loops:

- Growth model, new values depend on previous values

```
N = 20
for (i in 2:30) {
  f = rpois(1, 0.15*N[i-1])    # births
  d = rbinom(1, N[i-1], 0.1)   # deaths
  N[i] = N[i-1] + f - d
}

plot(seq_along(N), N)
```

Execute the code to see our simulated growth curve

## Problem: Growing objects is slow

```
N = 20
for (i in 2:30) {
  f = rpois(1, 0.15*N[i-1])    # births
  d = rbinom(1, N[i-1], 0.1)   # deaths
  N[i] = N[i-1] + f - d       # alive
}

plot(seq_along(N), N)
```

- Our growth model loop is slow because we are growing a vector at each iteration
- Solution pre-allocate vector (or any data type), then fill with loop

# Efficient memory usage

Improved code:

```
# Pre-allocate to correct size
N = vector(mode = "numeric", length = 30)
N[1] = 20
for (i in 2:30) {
  f = rpois(1, 0.15*N[i-1])    # births
  d = rbinom(1, N[i-1], 0.1)   # deaths
  N[i] = N[i-1] + f - d       # alive
}

plot(seq_along(N), N)
```

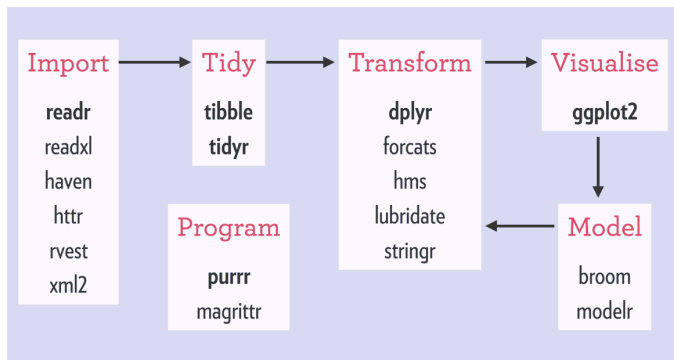
- In this simple example, does not make a difference
- With real-world, it will make a difference

# overview of loops

- Do not do things inside a loop that can be done outside
- Use vectorized functions whenever possible
- Map when function is not vectorized
  - Avoids the need to track indexes
  - Does the pre-allocation for you
- If you use loops, avoid growing lists/collections/data.frames
  - pre-allocate memory



## Brief introduction to programming with tidyverse



## Motivation:

- Analysts spend a lot of time manipulating and summarizing data
- Base R provides many functions for this, BUT:
  - the syntax is ugly
  - the functions can be slow/inefficient
- dplyr exists to make data manipulation easy to follow/correct/fast

# Install and load dplyr

- ggplot is included in the tidyverse package. To load the tidyverse package, run

```
library(tidyverse)
```

- If you get the message “there is no package ‘tidyverse’ ” you must install it first:

```
install.packages("tidyverse")  
library(tidyverse) # now load the tidyverse package
```

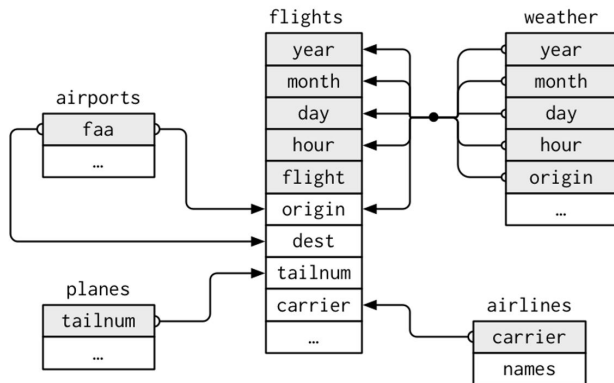
# Sample data set

- We will be using a data set containing all out-bound flights from NYC in 2013
- Available as an R package

```
#install.packages("nycflights13")  
library(nycflights13)  
#View(flights)
```

## nycflights13 tables

Multiple data sets in this “package” of data related to the NYCflights:



# “flights” table

Preview the flights data:

```
## # A tibble: 336,776 x 19
##   year month   day dep_time sched_dep_time dep_delay arr_time
##   <int> <int> <int>   <int>         <int>         <dbl>   <int>
## 1  2013     1     1     517             515           2     830
## 2  2013     1     1     533             529           4     850
## 3  2013     1     1     542             540           2     923
## 4  2013     1     1     544             545          -1    1004
## 5  2013     1     1     554             600          -6     812
## 6  2013     1     1     554             558          -4     740
## 7  2013     1     1     555             600          -5     913
## 8  2013     1     1     557             600          -3     709
## 9  2013     1     1     557             600          -3     838
## 10 2013     1     1     558             600          -2     753
## # ... with 336,766 more rows, and 12 more variables: sched_arr_time <int>,
## #   arr_delay <dbl>, carrier <chr>, flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dtm>
```

## dplyr verbs

dplyr provides a cogent, systematic way to carry out most data manipulation tasks:

These functions, called “verbs” are:

- `filter()` - keep rows matching desired properties
- `select()` - choose which columns you want to extract
- `arrange()` - sort rows
- `mutate()` - create new columns
- `summarize()` - collapse rows into summaries
- `group_by()` - operate on subsets of rows at a time

# dplyr verb properties

- Always take data set as the first parameter
- Returns a new data object, never updates/replaces original
- Specify columns as unquoted strings (symbols)
- use “pipes” (`%>%`) to pass result of one call on to the next
  - `%>%` operator passes the result of the left side to the first arg. on right
  - `a(b(c(x)))` equals `x %>% c() %>% b() %>% a()`

Example:

```
flights %>%                                # data  
  filter(carrier == "AA")                  # apply verb
```



## Filtering rows

- Find all flights to Detroit (DTW) in June (2013)
- Use `filter()` to complete the task

```
flights %>%                                # data  
  filter(dest == "DTW" & month == 6)      # filter DTW as dest.
```

Equivalently:

```
flights %>%                                # data  
  filter(dest == "DTW") %>%               # filter DTW as dest.  
  filter(month == 6)                      # On the DTW data, filter only
```

# Selecting columns

- To select specific columns, send a comma separated list of unquoted names

Select specific columns:

```
flights %>%  
  select(dep_time, arr_time, carrier)
```

Exclude specific columns:

```
flights %>%  
  select(-year, -tailnum)
```

Select range of columns:

```
flights %>%  
  select(month:dep_delay)
```

## Selecting columns: Part 2

```
flights %>% select(starts_with("d"))
```

```
flights %>% select(ends_with("time"))
```

```
flights %>% select(contains("arr"))
```

```
flights %>% select(-starts_with("d"))
```

See “?select” for complete list and examples

# Sort data

- Use `arrange()` to sort your rows

```
flights %>% arrange(sched_dep_time)
```

- Use `desc()` to reverse the sort order of a column

```
flights %>% arrange(month, desc(day))
```

- You can sort on functions of variables

```
flights %>% arrange(desc(dep_time - sched_dep_time))
```

# Create new variables

- Mutate allows you to create columns using existing values

```
flights %>%  
  mutate(speed = distance/(air_time/60)) %>%  
  arrange(desc(speed)) %>%  
  select(flight, speed)
```

- Remember, changes are not saved to “flights”, be sure to save the result if you want to use it later

```
new_flights <- flights %>% mutate(...)
```

# Use new variables right away

- The parameters to mutate are processed in the order they appear
  - You can use new variables right away

```
flights %>%  
  mutate( dist_km = distance * 1.61,  
           hours = air_time / 60,  
           kph = dist_km/hours ) %>%  
  select(flight, kph)
```

- Be careful! You can overwrite existing variables

# Summarize data

- You generally use `summarize()` to reduce the number of rows in your data by specifying summary functions for each of the columns

```
flights %>%  
  filter(!is.na(arr_delay)) %>%  
  summarize(avg_arr_delay = mean(arr_delay))
```

- Most useful summary functions:
  - `mean()`, `median()`, `var()`, `sd()`, `min()`, `max()`, `first()`, `last()`, `n()`, `n_distinct()`

# Summarize data

- Most useful summary functions:
  - `mean()`, `median()`, `var()`, `sd()`, `min()`, `max()`, `first()`, `last()`, `n()`, `n_distinct()`
- exercise: create one statement to calculate the mean and standard deviation for the departure delay (`dep_delay`) column



# Grouping data

- Often you want to perform summaries for groups of rows at a time
- `group_by()` function allow you to specify columns that define groups
- Functions like `mutate()` and `summarize()` are then performed for each group

## group\_by() + summarize() example

- Find the average arrival delay for each carrier, where all the missings are removed from arr\_delay

```
flights %>%  
  filter(!is.na(arr_delay)) %>%  
  group_by(carrier) %>%  
  summarize(avg_arr_delay = mean(arr_delay))
```

- Exercise: Improve the summary above to only include airlines with a negative mean arrival delay, i.e. the flights are early, on average

## Some special shortcuts

- `count()`
  - count number of rows with unique values of selected columns

```
flights %>% count(carrier)
```

- `summarize_all()/mutate_all()`
  - apply function to all columns

```
flights %>%  
  summarize_all(mean, na.rm=T)
```

- `summarize_at()/mutate_at()`
  - apply function to chosen columns

```
flights %>%  
  summarize_at(vars(ends_with("time")), mean, na.rm=T)
```

# Summarization exercises

- What month received the most number of flights to your home/favorite airport?
- What is the average airspeed for all flights?
- What was the average departure delay (for flights that actually had a departure)?
- What was the longest delay for each carrier (which carrier had the longest delay for a single flight)?

# Combining data frames

- `bind_rows()`
  - Stack two data frames on top of each other (should have the same number of columns)
- `bind_columns()`
  - Place two data frames next to each other (should have the same number of rows) – no merge-able columns
- `intersect()`, `union()`, `setdiff()`
  - For rows shared or exclusive to data frames

# ggplot2

- Even though the package is sometimes just referred to as “ggplot”, the package name is “ggplot2”
- ggplot is included in the tidyverse package. To load the tidyverse package, run

```
library(tidyverse)
```

- If you get the message “there is no package ‘tidyverse’ ” you must install it first:

```
install.packages("tidyverse")  
library(tidyverse) # now load the tidyverse package
```

# ggplot2 help

- Use the R help with

```
?ggplot
```

- Use the website: <http://ggplot2.tidyverse.org/reference/>
- Read Hadley's book (ggplot2: Elegant graphics for data analysis)

# Gapminder Data

- Dataset tracking life expectancy and per-capita GDP of 142 countries
- Data reported every five years from 1952-2007
- Data set is available in R package on CRAN:
  - `install.packages("gapminder")`

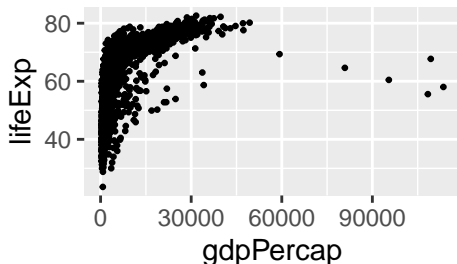
```
#install.packages("gapminder")  
library(gapminder)  
#View(gapminder)      # check out the data, scroll through it
```



# Create scatterplot

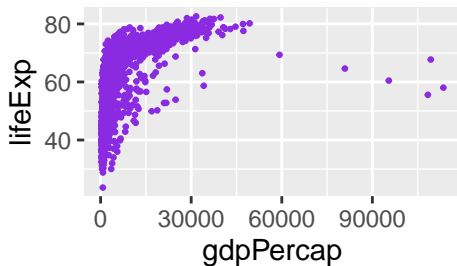
- Interest in how life expectancy relates to GDP per capita
  - Does life expectancy increase with per capita GDP?
- Create a scatterplot of gdpPercap (Y-axis) vs lifeExp (X-axis)

```
ggplot(data = gapminder, aes(x=gdpPercap, y=lifeExp)) +  
  geom_point(size = 0.5)
```



## Add some color

```
ggplot(data = gapminder, aes(x=gdpPercap, y=lifeExp)) +  
  geom_point(size = 0.5, color = "blueviolet")
```



- Check out the color names that R knows:

```
colors()
```

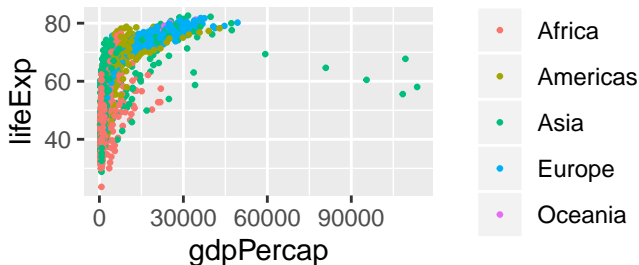
- Can also take HEX values, e.g. "#8A2BE2"

# Color by the data

Incorporate color into the plot in a useful way:

- Color by continent to see trends by region of the world

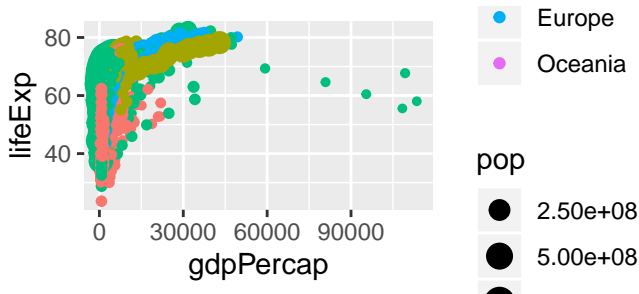
```
ggplot(data = gapminder, aes(x=gdpPercap, y=lifeExp, color = continent)) +  
  geom_point(size = 0.5)
```



# Scale point size by data

- Make countries with larger populations have larger dots
  - How big are the high GDP countries?

```
ggplot(data = gapminder, aes(x=gdpPercap, y=lifeExp, color = continent, size = pop)) +  
  geom_point()
```

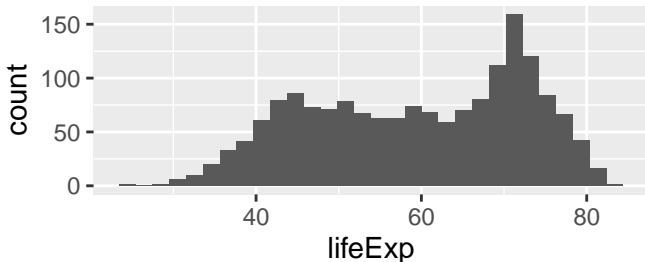


# Geometries

- `geom_point()` is just one of many geometries:
  - Used to make scatter plots
  - Works best with two continuous variables
- What if we wanted to look at a distribution of a single continuous variable?

```
ggplot(data = gapminder, aes(x=lifeExp)) +  
  geom_histogram() # function for histograms
```

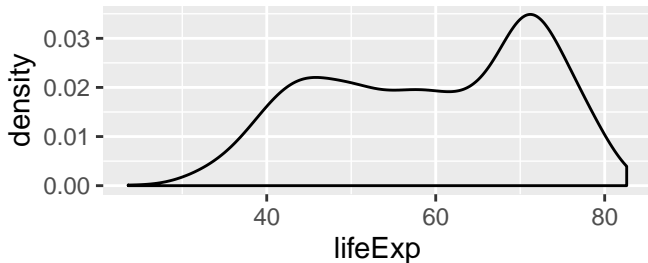
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



# Geometries

- `geom_point()` is just one of many geometries:
  - Used to make scatter plots
  - Works best with two continuous variables
- What if we wanted to look at a distribution of a single continuous variable?

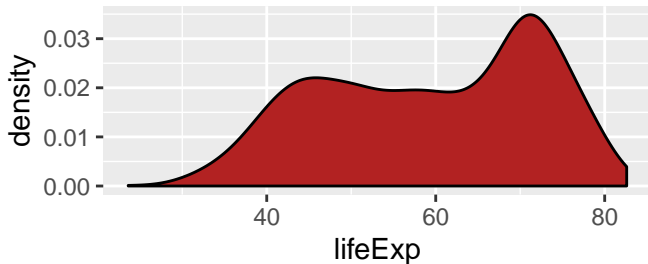
```
ggplot(data = gapminder, aes(x=lifeExp)) +  
  geom_density() # function for smoothed density
```



# Density plot with custom aesthetics

- Add color fill

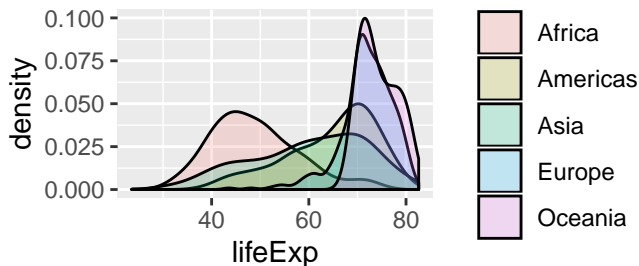
```
ggplot(data = gapminder, aes(x=lifeExp)) +  
  geom_density(fill = "firebrick") # function for smoothed density
```



# Single variable across groups

- Plot density over life expectancies by continent
  - Fill continents with their own color to distinguish them

```
ggplot(data = gapminder, aes(x=lifeExp, fill = continent)) +  
  geom_density(alpha = 0.2)  # function for smoothed density
```



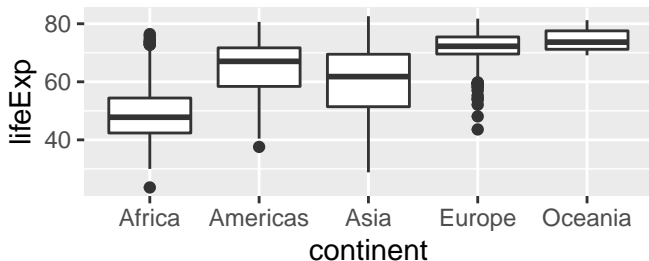


# Single variable across groups

- Another approach to the same problem:
  - Use boxplots to characterize differences across groups

*# Notice how aes change*

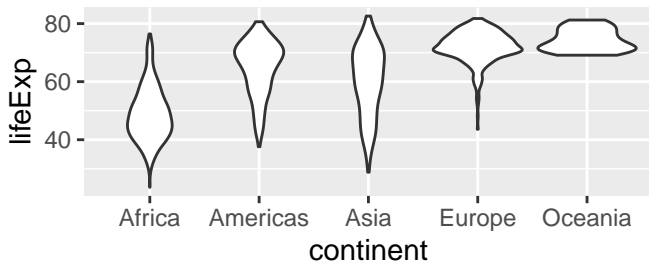
```
ggplot(data = gapminder, aes(x=continent, y = lifeExp)) +  
  geom_boxplot()
```



# Single variable across groups

- Violin plot:

```
# Notice how aes change  
ggplot(data = gapminder, aes(x=continent, y = lifeExp)) +  
  geom_violin()
```

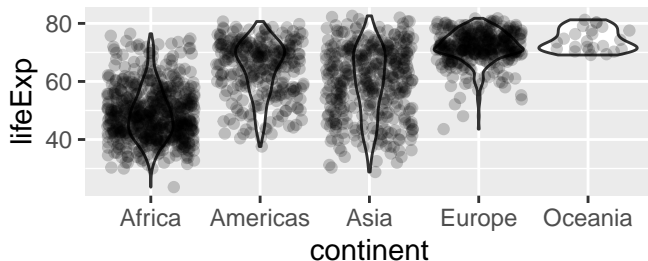


# Stack geometries in layers

- How can we display the plot with data overlay?

*# Notice how aes change*

```
ggplot(data = gapminder, aes(x=continent, y = lifeExp)) +  
  geom_violin() +  
  geom_jitter(alpha = 0.2) # jitter the points
```



## Day 3

If you were not present for Days 1 and 2

- Go to <https://github.com/mkleinsa/Rbootcamp>
- Click Clone or download
- Download .zip archive
- extract to desktop, open slides/slides.Rmd to follow along
- Cntrl + f search “Day 3” to find today’s slides

# Simulation Basics

Why simulate?

- confirm model/method works by mimicking the real world
- Get intuition before writing a proof
- \*Closed form solution does not exist, only option to simulate

Challenge in figuring out how to model real world processes with numbers and variables

Note: randomness in R is pseudo-random. The default random number generator depends on a seed which is some initial starting value where random numbers grow from. Always set a seed.

# Simulation Basics

- Often interested in expected values of random variables or probability of events occurring
- Through simulation, we don't calculate these values directly.

Instead:

- 1 Create a sample with the given constraints
- 2 From sample, calculate the mean over the random draws to estimate expected value, or, calculate an observed frequency to estimate the probability.

# Simulation Basics

Sampling in R from sets:

- Use the `sample()` function to sample from a discrete set

Example - Simulate a coin toss experiment:

- toss a coin ten times, record what side it lands on each time

```
set.seed(7794)
sample(c("H","T"), size = 10, replace = TRUE) # fair coin
```

```
## [1] "H" "H" "H" "H" "T" "T" "T" "T" "T" "H"
```

```
sample(c("H","T"), size = 10, replace = TRUE,
       prob = c(0.6, 0.4)) # weighted coin
```

```
## [1] "H" "H" "H" "T" "H" "T" "T" "H" "T" "H"
```

# Simulation Basics

Sampling in R from distributions:

- `d____(x)` returns the density of a probability distribution for a discrete value of  $x$
- `p____(q)` returns the cumulative density function (CDF) up to  $q$
- `q____(p)` returns the quantile from a cumulative probability
- `r____(n)` returns a random deviate (a simulation of random draw) of size  $n$

Example:

```
qnorm(0.025); dnorm(0)
```

```
## [1] -1.959964
```

```
## [1] 0.3989423
```

?distributions for more information



# Simulation

distribution	function
Normal	<code>norm(n, mean=0, sd=1)</code>
exponential	<code>exp(n, rate=1)</code>
uniform	<code>unif(n, min=0, max=1)</code>
gamma	<code>gamma(n, shape, scale=1)</code>
poisson	<code>pois(n, lambda)</code>
Weibull	<code>weibull(n, shape, scale=1)</code>
Cauchy	<code>cauchy(n, location=0, scale=1)</code>
beta	<code>beta(n, shape1, shape2)</code>
Student t	<code>t(n, df)</code>
binomial	<code>binom(n, size, prob)</code>
logistic	<code>logis(n, location=0, scale=1)</code>

**Table 3:** Built-in distributions

# Simulation: sample from probability distributions

```
rnorm(6) # 6 standard normal deviates  
rnorm(10, mean=50, sd=19) # set mean and spread  
runif(10, min=0, max=1) # uniform distribution  
rpois(10, lambda=15) # Poisson
```

Cointoss reframed: toss coin 8 times using binomial distribution

```
set.seed(7794)  
rbinom(8, size=1, p=0.5) # 8 coin tosses
```

# Some simple simulations

Problem: predict the number of girls in 400 births in a population where prob. of female birth is 48.8%

```
#set.seed()  
n.girls = rbinom(n=1, size=400, prob=0.488)  
n.girls
```

```
## [1] 199
```

- Get a distribution of the number of female births when prob. = 0.488

```
n.sims=1000  
n.girls=rbinom(n.sims,400,0.488)  
hist(n.girls)
```

# Some simple simulations

- Exercise: calculate the expected number of girls born.

```
# estimate expected value by computing the mean over all  
# draws  
sum(n.girls)/n.sims
```

- Does your estimate of the expected number of girls born make sense?  
Check:

```
# expected value of a binomial r.v. is n*p  
n.sims*0.488
```

- Exercise: calculate the probability that the number of girls born will exceed half the total population size.

```
# estimate the probability by calculating the desired  
# frequency  
sum(n.girls > 200)/n.sims
```

# Replicate for repeating simulations

Function to simulate the mean of 100 standard normal draws:

```
sim.mean = function() { mean(rnorm(100)) }
```

The following for loop repeats the simulation 1,000 times:

```
sims = vector(mode = "numeric", length = 1000L)
for(i in 1:1000) { sims[i] = sim.mean() }
```

We can avoid the for loop with **replicate()**:

```
sims = replicate(1000, sim.mean())
```

# Replicate example

- Recommendation: for each solution you produce,

```
sim = replicate(1000,
  {
    insert simulation code here
  })
```

Example: use the `sample()` function to draw a collection of M/F values where the probability of F is 0.75. What is the probability of getting more males than females in a set of 10 random people?

```
experiment = replicate(1000, {
  mydraw = sample(c("M","F"), 10, replace = T,
    prob = c(0.25,0.75))
  sum(mydraw=="M") > sum(mydraw == "F")
})

mean(experiment)
```

# Simulation: Confidence intervals

Open simulations.R script in course files

Exercise: Is the coverage rate of confidence intervals for the mean accurate?

- Given conditions similar to ours, does our method for generating confidence intervals capture the true value 95% of the time?

