Introduction, R and `ggplot2`

STAT 135: Concepts of Statistics

GSI: Andy Shen

Acknowledgments: Chris Paciorek, Michael Tsiang

UC Berkeley, Fall 2022

Section 1

Highlights from the Syllabus

Logistics

- GSI: Andy Shen
- Email: aashen@berkeley.edu
- Class Email: stat135-staff@lists.berkeley.edu
- You may email me with any administrative/private questions or concerns (please include **Stat 135 102/105** in the subject).
- Course announcements will be sent to your email via bCourses (Canvas). Please check your emails regularly for such announcements.
- Any questions about course content (lectures, homework, exams) should be posted **on Piazza** (join code XYZ)

Lab Sections

- **Lab 102:** 11:00am - 1:00pm in Evans Hall 342
- **Lab 105:** 3:00 - 5:00pm in Evans Hall 332

To get credit for in-class quizzes, you must attend the section you are registered for.

Waitlisted students

If you are on the wait list and would like to class, please let me know by staying after class or sending me an email. You may be able to enroll in a different lab section.

Office Hours

- Mondays: 1:00 - 2:00 PM
- Thursdays: 10:30 - 11:30 AM
- All office hours will be in **Evans Hall 345**
- These are still subject to change!

Homework

- HW1 is posted on bCourses and is due **next Friday, Sept 2**
- Homework will be assigned weekly for a total of 10 assignments
- Late assignments will be accepted with a 50% penalty **until solutions are posted.** HW submitted after that will receive no credit.
- Please submit homework to Gradescope. **You must tag your questions correctly.** Improperly tagged questions will not receive credit since the grader cannot see it!
- Is everyone registered on Gradescope?

Quizzes

There will be four 50 minute quizzes during lab sessions to test your understanding of most recent lectures and homeworks. The dates of the quizzes are:

Quiz 1: September 9

Quiz 2: September 23

Quiz 3: November 4

Quiz 4: November 18

We will drop the lowest quiz score from the final grade. For this reason, we will not accommodate make-up quizzes unless under unusual and unexpected circumstances.

Exams

There will be one midterm, and a three hour final exam. The exams will be open-book, open-note. The midterm will take place during class. You must take the exams to pass the class.

The test dates are

- Midterm: Wednesday October 12, during class
- Final: Currently scheduled for Monday December 12, 8AM-11AM. Location TBD

DSP Accommodations

If you require DSP accommodations, email both myself and Prof. Jahani and send us the letter through the DSP portal **before September 5!** We cannot provide you with the proper accommodations unless we receive the letter. DSP will email about this in the beginning of the semester.

Section 2

Review of R Programming

Overview

- Base R commands
- Good for fast and simple tasks
- Easy to learn, no need to install packages
- `tidyverse` commands
  - Good for more complex tasks
  - You can generally do more with tidyverse commands
  - Syntax is slightly different from Base R (but not much)
  - `dplyr`: data manipulation and wrangling
  - `ggplot2`: making graphics (looks much nicer than Base R)
  - `tidyr`: reshaping data frames
  - . . . and much more!

This class

- You can most likely stick to Base R for most of your commands
- For plotting, I suggest using `ggplot2` because the plots are generally easier to see and interpret but it is up to you. Just make sure your plots are readable.

Section 3

Base R

Object assignment

Use <- to assign an object to a value (or an existing object).

```
x <- 2
y <- x + 3
```

```
x
```

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

```
y
```

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

Data Structures

R has a wide variety of data types including *scalars*, *vectors* (numerical, character, logical), *matrices*, *data frames*, and *lists*.

For example the number 2 is a scalar. We can give 2 the name a using the assignment operator.

```
a <- 2  #assign the value 2 to a
a
```

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

Similarly we can assign the name vec to a character vector as follows:

```
vec <- c("I","love","learning","R")
vec
```

```
## [1] "I"        "love"     "learning" "R"
```

Here the values I, and love are strings which we must put in quotes so that R doesn't think they are the name for something.

Vectors

- Vectors are the most important family in R. They form the basis for matrices, data frames, and linear algebra computations
- Create a vector using `c()`
- A string of consecutive integers can be created using the colon operator: `:`

```
x <- c(1, 2, 3)
x
```

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

```
1:9
```

```
## [1] 1 2 3 4 5 6 7 8 9
```

```
rep(9, 4)
```

```
## [1] 9 9 9 9
```

```
rep(c(1, 2), 3)
```

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

Vector Recycling

When applying arithmetic operations to two vectors of different lengths, R will automatically recycle, or repeat, the shorter vector until it is long enough to match the longer vector.

Question: What is the output of the following commands?

```
c(1, 3, 5) + c(5, 7, 0, 2, 9, 11)
c(1, 3, 5) + c(5, 7, 0, 2, 9)
```

```r
c(1, 3, 5) + c(5, 7, 0, 2, 9, 11)
```

```
## [1]  6 10  5  3 12 16
```

```r
c(1, 3, 5) + c(5, 7, 0, 2, 9)
```

```
## [1]  6 10  5  3 12
```

Matrices

- A **matrix** in R is a vector, but with a dimension attribute of length 2 (rows and columns)
- You can create a matrix using `matrix()`, `rbind()` or `cbind()`

```
M1 <- matrix(1:9, nrow = 3, ncol = 3)
M2 <- cbind(c(1, 2, 3), c(4, 5, 6), c(7, 8, 9))
M3 <- rbind(c(1, 4, 7), c(2, 5, 8), c(3, 6, 9))
```

```
all(M1 == M2)
```

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

```
all(M1 == M3)
```

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

You can also create a matrix row-wise using `byrow = TRUE` in the `matrix()` command:

```
matrix(1:9, nrow = 3, ncol = 3, byrow = TRUE)
```

```
##      [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
```

R Functions

Functions are a special type of object in R.

A function in R takes in certain input objects called **arguments** and returns an output by executing a set of commands. The basic syntax to create your own function is:

```
functionName <- function(arg_1, arg_2, ...) {
  # the body of the function goes here
}
```

Once a function is created, a call to the function is then implemented as:

```
functionName(expr1, expr2, ...)
```

Flow Control: the `if()` statement

An `if()` statement evaluates conditions inside the statement, and if `TRUE`, executes the commands inside:

```
x <- 3
if (x > 1) {
  print("x is greater than 1.")
}
```

```
## [1] "x is greater than 1."
```

Keep in mind the statement inside the parenthesis must evaluate to `TRUE`.

if-else

If there is an alternative set of commands to run when the `if()` condition is `FALSE`, you can enclose them in an `else` statement:

```
x <- 3
if (x > 1) {
  print("x is greater than 1.")
} else {
  print("x is not greater than 1.")
}
```

```
## [1] "x is greater than 1."
```

Repeated execution: `for()` loops

The `for()` loop is used for repeating an execution a fixed number of times.

```
v <- 1:9
out <- c()
for (i in v) {
  if (i %% 2 == 0) {
    out <- c(out, "even")
  } else {
    out <- c(out, "odd")
  }
}
```

```
v
```

```
## [1] 1 2 3 4 5 6 7 8 9
```

The previous block of code is inefficient and bad practice. You should not incrementally grow a vector in R. In Python it is okay to do `list.append`, but not in R.

The reason for this is because R has to recreate the vector every time it is re-assigned. For longer vectors, this will be very time-consuming. For certain tasks, this is unavoidable, but you should be efficient as often as possible.

A more efficient version:

```
v <- 1:9
out <- rep(NA, length(v))
for (i in v) {
  if (i %% 2 == 0) {
    out[i] <- "even"
  } else {
    out[i] <- "odd"
  }
}
```

```
out
```

```
## [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"
```

Here we pre-define the `out` vector as a length 9 vector of `NA`s. When R goes through the loop, it can just replace theith value of `out` with `"even"` or `"odd"`.

ifelse()

A vectorized version of the `if-else` statement is the `ifelse()` function.

The `ifelse()` function has the form

```
ifelse(condition_vector, vector_1, vector_2)
```

```
ifelse(1:9 %% 2 == 0, "even", "odd")
```

```
## [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"
```

This faster than using a `for()` loop.

Repeated execution: `while()` loops

The `while()` statement creates a loop that repeats a set of commands for as long as a certain condition holds:

```
i <- 0
while (i < 6) {
  print(paste0("i is ", i)) # paste0 concatenates objects and character strin
  i <- i + 1
}
```

```
## [1] "i is 0"
## [1] "i is 1"
## [1] "i is 2"
## [1] "i is 3"
## [1] "i is 4"
## [1] "i is 5"
```

Question: what is the value if `i`?

```
i
```

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

Section 4

Simulation in R

Parameter Estimation

- Imagine that we have a population and we are interested in some characteristics of that population
  - Example: the sleep time of all UC Berkeley undergraduates majoring in statistics
- In a perfect world, we may want to perform a census and calculate the characteristic (or parameter) of interest.
- However, it is unfeasible and sometimes impossible to find this parameter of interest. As a result, we obtain a subset of this population (a sample), and under certain sampling conditions, we can obtain estimates these characteristics.

Parameters, Statistics, Estimators

- A **parameter** is some constant (usually unknown) that is a characteristic of the population.
- A **statistic** is a random variable that is a function of the observed data.
  - It is important to note that statistics *are not* a function of the parameter of interest.
- An **estimator** is a statistic related to some quantity of the population characteristic.
- A **sampling distribution** is the probability distribution of a statistic.

Simulation in R

- You can simulate random numbers using R
- This will come in handy for visualizing how certain distributions converge as you increase the number of samples
- Simulation is done by using pseudorandom number generation: computers follow a deterministic algorithm to generate numbers. It is not truly "random"
- However, we don't know or care about what this algorithm is, so the numbers appear random to us
- The user can specify (or set) the seed using the `set.seed()` function so that the "random" numbers are the same every time the code is run
- Being able to set the seed allows researchers to reproduce simulation results

The sample() function

- Samples a given numbers from a vector with or without replacement (default without replacement).

```
set.seed(135)
sample(5) # same as sample(1:5) but may yield different numbers
```

```
## [1] 1 2 5 4 3
```

```
sample(5, replace = TRUE)
```

```
## [1] 5 5 5 4 5
```

```
sample(1:10, size = 2, replace = TRUE)
```

```
## [1] 7 5
```

A fair coin:

```
sample(c("Heads", "Tails"), size = 6, replace = TRUE, prob = rep(0.5, 2))
```

```
## [1] "Tails" "Tails" "Heads" "Heads" "Heads" "Tails"
```

By default, the `prob` argument gives equal probability to each entry in the vector.

A loaded coin:

```
sample(c("Heads", "Tails"), size = 6, replace = TRUE, prob = c(0.9, 0.2))
```

```
## [1] "Tails" "Heads" "Heads" "Heads" "Heads" "Heads"
```

Section 5

Simulating from Probability Models

Probability Models

A **probability model** is a description of how we think data is generated. A probability model is not actually how the data is generated (this is why it is called a model), but it is used as a reasonable description for how the data appears to vary. As analysts, we never get to see the true underlying generation process, so we need to estimate it as best we can. That is the purpose of this class.

Random Variables

A **random variable** is set of possible values that come from some random phenomenon. A **discrete random variable** takes on a countable number of distinct values (such as the number of heads found in n trials). A **continuous random variable** can take on an infinite number of values in an interval (such as the amount of water in a lake).

The **probability mass function (PMF)** $P(X = x)$ is the probability that a *discrete random variable* takes on a specific value. We refer to the support of a probability mass function as the set of values that the discrete random variable takes on.

Some properties of the PMF of a random variable include:

- $\mathbb{P}(X = x) \geq 0$
- $\sum_x P(X = x) = 1$ (discrete)

The **probability density function (PDF)** $f(x)$ is the probability that a continuous random variable takes on a specific range.

Similarly, some properties of the PDF of a random variable include:

- $f(x) \geq 0$
- $\int_{-\infty}^{\infty} f(x)dx = 1$

The **cumulative distribution function (CDF)** of a random variable, X is defined to be:

$$F_X(x) = P(X \leq x);\ x \in (-\infty, \infty)$$

Some properties of the **CDF** of a random variable include:

- $\lim_{x \to -\infty} F(x) = 0$
- $\lim_{x \to \infty} F(x) = 1$
- CDF is non-decreasing (if $x \leq y$, then $F(x) \leq F(y)$) and right-continuous.

Section 6

Simulating in R

Random Number Generation

For simplicity, let's illustrate using the normal distribution.

`rnorm()` generates random numbers from the normal distribution (you can specify μ and σ). Be default, $\mu = 0$ and $\sigma = 1$ (standard normal).

```
rnorm(9)
```

```
## [1] -1.0671947  0.3016295 -1.4654787 -0.3800575 -0.2321954 -0.7244148  0.36
## [8]  0.8632986 -0.5540962
```

```
rnorm(9, mean = 12, sd = 3)
```

```
## [1]  5.913963 13.569024  9.556523 12.489977 12.766649 11.963748 12.662282
## [8] 10.830564 15.466817
```

Probability Density Functions

Recall the probability density function for a normal random variable X is given by:

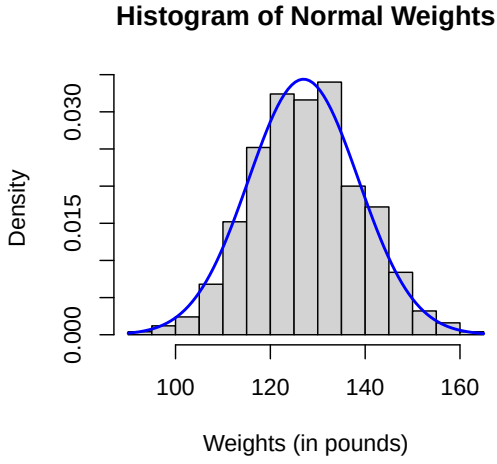
$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

- The `dnorm()` function evaluates the normal density function. The syntax for the main arguments of `dnorm()` is `dnorm(x, mean, sd)`. The function inputs a numeric value (or vector) in the `x` argument and computes the value of the normal density function at `x` when the μ is `mean` and σ is `sd`.
- The `dnorm()` function can be used to visualize the normal density curve. It is particularly useful to superimpose the normal density curve over a histogram of observed values from a normal distribution.

Illustration

Suppose we simulate drawing a sample of 500 weights from a population that follows a normal distribution with a mean weight of 127 pounds and a standard deviation of 11.6 pounds. We can plot the histogram of the weights in the sample and superimpose the normal density curve over it:

```
# Set the seed for reproduceability
set.seed(321)
# Sample 500 random numbers from N(127,11.6).
weights <- rnorm(500, 127, 11.6)
# Plot the weights on a histogram.
hist(weights, prob = TRUE, xlab = "Weights (in pounds)",
     main = "Histogram of Normal Weights")
# Add the N(127,11.6) density curve.
curve(dnorm(x, 127, 11.6), lwd = 2, col = "blue", add = TRUE)
```


Histogram of Normal Weights
Density
0.000
0.015
0.030
100
120
140
160
Weights (in pounds)

Cumulative Distribution Functions

- The `pnorm()` function is the distribution function for normal random variables. More simply put, `pnorm()` computes probabilities from a normal distribution with specified mean and standard deviation.

The syntax for the main arguments of `pnorm()` is `pnorm(q, mean, sd, lower.tail)`.

The function inputs a numeric value (or vector) in the q argument and computes the probability that a value drawn from a normal distribution with μ `mean` and σ `sd` will be less than or equal to q. Simply put, it takes in a z-score and returns a probability.

```
pnorm(1.96)
```

```
## [1] 0.9750021
```

The optional argument `lower.tail` inputs a logical value and changes the direction of the probability. The default is `lower.tail = TRUE`, so the `pnorm()` function will compute the probability that a value drawn from the normal distribution will be less than or equal to q.

If we set `lower.tail = FALSE`, then the `pnorm()` function will compute the probability that a value drawn from the normal distribution will be greater than or equal to q.

```
pnorm(1.96, lower.tail = FALSE)
```

```
## [1] 0.0249979
```

Quantiles

The `qnorm()` function is the quantile function for the normal distribution.

The syntax for the main arguments of `qnorm()` is `qnorm(p, mean, sd)`. The function inputs a probability value (or vector) in the `p` argument and computes the quantile for that probability from a normal distribution with μ `mean` and σ `sd`.

Simply put, `qnorm()` accepts a probability and returns a z-score:

```
qnorm(0.975)
```

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

Notice how `qnorm()` and `pnorm()` are opposites?

```
qnorm(pnorm(0.97))
```

```
## [1] 0.97
```

Simulating from other distributions

We are not just limited to sampling from normal distributions. We can use the r, d, p, and q prefixes to most known probability distributions.

Examples:

- `runif`: generate random numbers from uniform distribution
- `dpois`: density function for Poisson distribution
- `pbinom`: distribution function for binomial distribution
- `qt`: quantile function for t-distribution

Section 7

Graphics: `ggplot2`

Graphics: ggplot2

```
library(ggplot2)
library(dplyr) # imports the pipe operator %>%
```

```
##
## Attaching package: 'dplyr'
```

```
## The following objects are masked from 'package:stats':
##
##     filter, lag
```

```
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
library(gapminder) # imports our dataset
```

The pipe operator: %>%

The pipe operator is a tidyverse command that allows for more streamlined function calls.

Say I have a function `f()`. The command `x %>% f()` is equivalent to `f(x)`. The object that gets piped into the function is always the first argument of the function.

```
x <- 1:3
mean(x)
```

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

```
x %>% mean()
```

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

Pros/cons of base graphics, ggplot2, and lattice

Base graphics is

- good for exploratory data analysis and sanity checks
- inconsistent in syntax across functions: some take x,y while others take formulas
- defaults plotting parameters are ugly, and it can be difficult to customize
- that said, one can do essentially anything in base graphics with some work

`ggplot2` is

- generally more elegant
- more syntactically logical (and therefore simpler, once you learn it)
- better at grouping
- able to interface with maps

We'll focus on `ggplot2` as it is very powerful, very widely-used and allows one to produce very nice-looking graphics without a lot of coding.

Grammar of graphics

`ggplot2` syntax is very different from base graphics and lattice. It's built on the **grammar of graphics**. The basic idea is that the visualization of all data requires four items:

1. One or more **statistics** conveying information about the data (identities, means, medians, etc.)
2. A **coordinate system** that differentiates between the intersections of statistics (at most two for ggplot, three for `lattice`)
3. **Geometries** that differentiate between off-coordinate variation in *kind*
4. **Scales** that differentiate between off-coordinate variation in *degree*

`ggplot2` allows the user to manipulate all four of these items through the `stat_*`, `coord_*`, `geom_*`, and `scale_*` functions.

All of these are important to *truly* becoming a `ggplot2` master, but today we are going to focus on the most important to basic users and their data layers: `ggplot2`'s *geometries*.

Basic Usage: ggplot2

The general call for ggplot2 graphics looks something like this:

```
# NOT run
ggplot(data = df, aes(x = , y = , [options])) + geom_xxxx() + ... + ... + ...
```

OR, with %>%:

```
df %>% ggplot(aes(x = , y = , [options])) + geom_xxxx() + ... + ... + ...
```

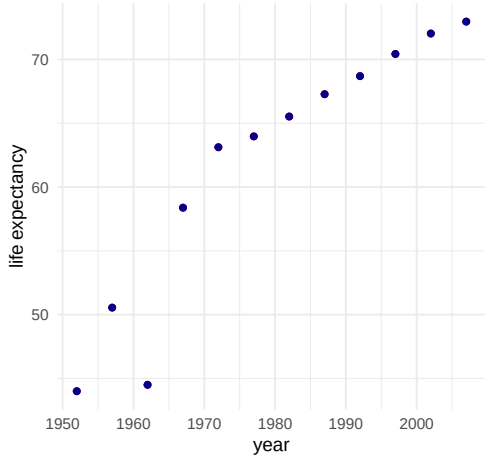
Note that ggplot2 graphs in layers in a *continuing call* (hence the endless $+\ldots+\ldots+\ldots$), which really makes the extra layer part of the call.

```
... + geom_xxxx(data = , aes(x = , y = ,[options]), [options]) + ... + ... +
```

You can see the layering effect by comparing the same graph with different colors for each layer

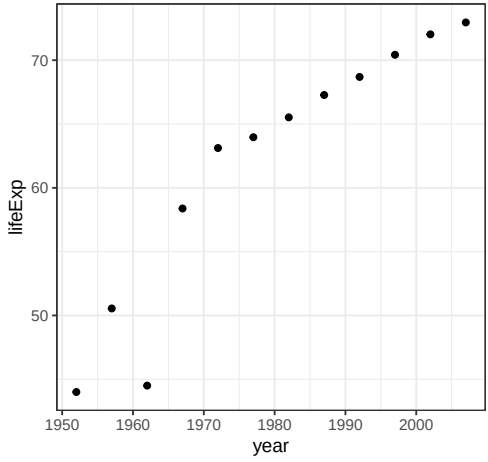
```
gapChina <- gapminder %>% filter(country == "China")
p <- ggplot(data = gapChina, aes(x = year, y = lifeExp)) +
                  geom_point(color = "red")
```

```
p + geom_point(aes(x = year, y = lifeExp), color = "darkblue") +
  ylab("life expectancy") +
    theme_minimal()
```

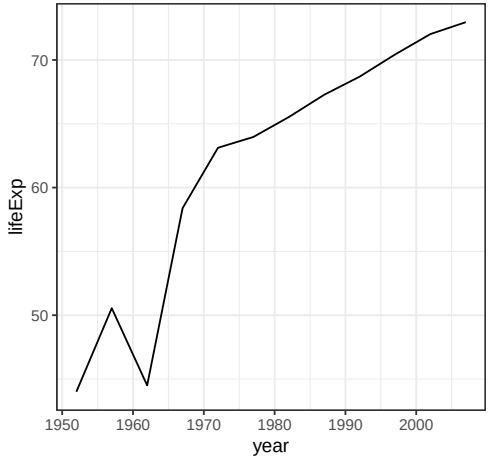


And, if you're desperate for the quick and dirty functionality of base plot, or just like the more familiar syntax at first, ggplot2 offers the `qplot()` function as a wrapper for most basic plots:

```
qplot(x = year, y = lifeExp, data = gapChina) + theme_bw()
```



```
qplot(x = year, y = lifeExp, data = gapChina, geom = "line") + theme_bw()
```



A basic walkthrough of ggplot2

See R script!

Expected Value

The **expectation of a discrete random variable** X with probability mass function $(P(X = x))$ is:

$$\mathbb{E}[X] = \sum_x x \ \mathbb{P}(X = x)$$

The **expectation of a continuous random variable** X with probability density function $(f(x))$ is:

$$\mathbb{E}[X] = \int_x x f(x) dx$$

Variance

The **variance of a random variable** X is the expectation of the squared deviation of the random variable from its mean.

$$\mathsf{Var}(X) = \mathbb{E}[(X - \mathbb{E}(X))^2] = \mathbb{E}(X^2) - [\mathbb{E}(X)]^2$$

Exercise

Show that the expectation and variance of a Gamma(α, β) distribution is $\frac{\alpha}{\beta}$ and $\frac{\alpha}{\beta^2}$, respectively.

The pdf of the Gamma(α, β) distribution is:

$$f(x) = \frac{\beta^\alpha}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x}, \qquad \text{for } x > 0, \quad \alpha, \beta > 0.$$

Hint: $\Gamma(\alpha) = \frac{\Gamma(\alpha+1)}{\alpha}$.

Solution