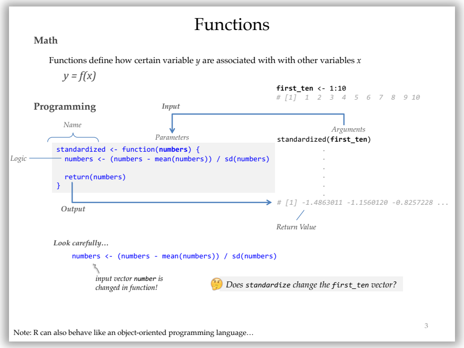
Business Analytics Using Computational Statistics

No Class

Week 4
Computational Intervals

Week 5
Bootstrapping

Functions and Loops



Functions

Math

Functions define how certain variable y are associated with with other variables x

$y = f(x)$

Programming

```
first_ten <- 1:10
# [1]  1  2  3  4  5  6  7  8  9 10
```

Input

Name

Parameters

Arguments

```
standardized <- function(numbers) {
  numbers <- (numbers - mean(numbers)) / sd(numbers)

  return(numbers)
}
```

```
standardized(first_ten)
# [1] -1.4863011 -1.1560120 -0.8257228 ...
```

Logic

Output

Return Value

Look carefully…

```
numbers <- (numbers - mean(numbers)) / sd(numbers)
```

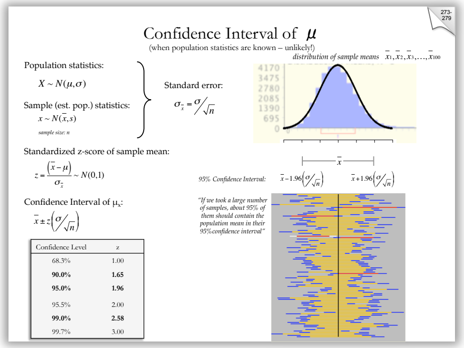
input vector `number` is changed in function!

🤔 *Does `standardize` change the `first_ten` vector?*

Note: R can also behave like an object-oriented programming language…

3

Confidence Intervals



273-279

Confidence Interval of μ

(when population statistics are known – unlikely!)

Population statistics:

$X \sim N(\mu,\sigma)$

Sample (est. pop.) statistics:

$x \sim N(\bar{x},s)$

sample size: n

Standard error:

$\sigma_{\bar{x}} = \sigma / \sqrt{n}$

distribution of sample means $\bar{x}_1, \bar{x}_2, \bar{x}_3, \ldots, \bar{x}_{100}$

Standardized z-score of sample mean:

$$z = \frac{(\bar{x}-\mu)}{\sigma_{\bar{x}}} \sim N(0,1)$$

95% Confidence Interval: $\bar{x} - 1.96\left(\sigma/\sqrt{n}\right)$ $\bar{x} + 1.96\left(\sigma/\sqrt{n}\right)$

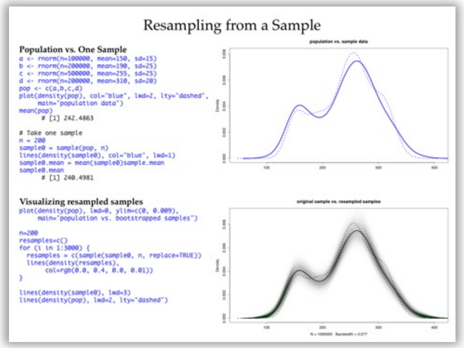
Confidence Interval of μ_x:

$$\bar{x} \pm z\left(\sigma/\sqrt{n}\right)$$

"If we took a large number of samples, about 95% of them should contain the population mean in their 95%confidence interval"

Confidence Level	z
68.3%	1.00
90.0%	**1.65**
95.0%	**1.96**
95.5%	2.00
99.0%	**2.58**
99.7%	3.00

Bootstrapping



Resampling from a Sample

Population vs. One Sample

```
a <- rnorm(n=100000, mean=150, sd=15)
b <- rnorm(n=200000, mean=190, sd=25)
c <- rnorm(n=500000, mean=255, sd=25)
d <- rnorm(n=200000, mean=310, sd=20)
pop <- c(a,b,c,d)
plot(density(pop), col="blue", lwd=2, lty="dashed",
     main="population data")
mean(pop)
      # [1] 242.4863

# Take one sample
n = 200
sample0 = sample(pop, n)
lines(density(sample0), col="blue", lwd=1)
sample0.mean = mean(sample0)sample.mean
sample0.mean
      # [1] 240.4981
```

Visualizing resampled samples

```
plot(density(pop), lwd=0, ylim=c(0, 0.009),
     main="population vs. bootstrapped samples")

n=200
resamples=c()
for (i in 1:3000) {
  resamples = c(sample(sample0, n, replace=TRUE))
  lines(density(resamples),
        col=rgb(0.0, 0.4, 0.0, 0.01))
}

lines(density(sample0), lwd=3)
lines(density(pop), lwd=2, lty="dashed")
```

population vs. sample data
Density
100
200
300
400
original sample vs. resampled samples
N = 1000000 Bandwidth = 3.077

R Packages

Collections of functions for you to install, load, and use freely

Install from CRAN (Comprehensive R Archive Network)

Mature and well tested packages are usually available on a central repository – CRAN

```
install.packages("ggplot2")

library("ggplot2")
# Ready to use all ggplot2 functions
```

Install from Github

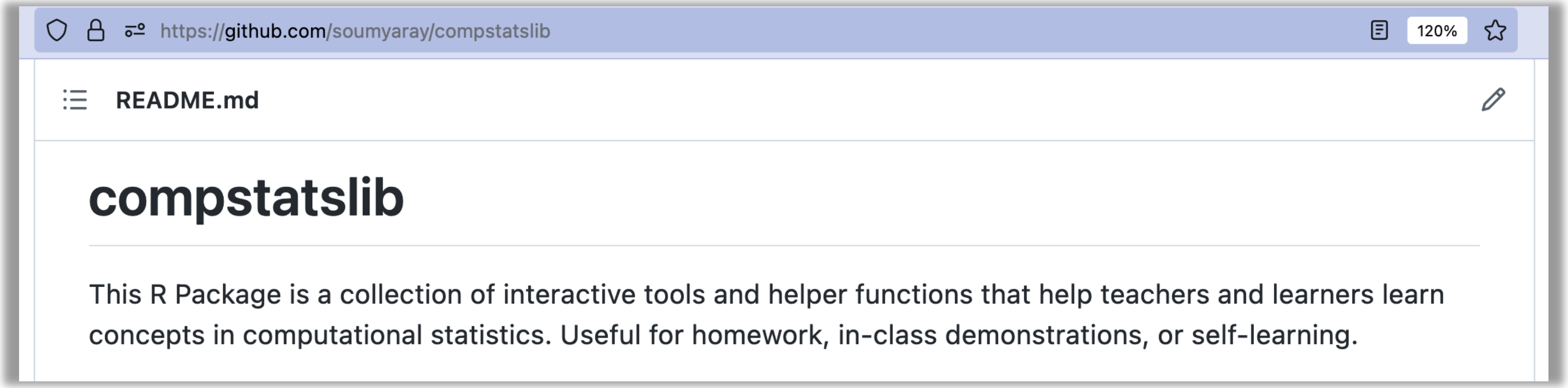
New or experiemtal packages are usually available on on Github – where developers self-publish their own code

```
install.packages("remotes")
remotes::install_github("soumyaray/compstatslib")

library("compstatslib")
# Ready to use all compstatslib functions

machine_precision()
[1] 2.220446e-16
```

`machine_precision()` reports the smallest possible number on your machine such that `1 + x != 1`

Soumya Ray *Daniele Melotti*

National Tsing Hua University



https://github.com/soumyaray/compstatslib

README.md

compstatslib

This R Package is a collection of interactive tools and helper functions that help teachers and learners learn concepts in computational statistics. Useful for homework, in-class demonstrations, or self-learning.

https://github.com/soumyaray/compstatslib

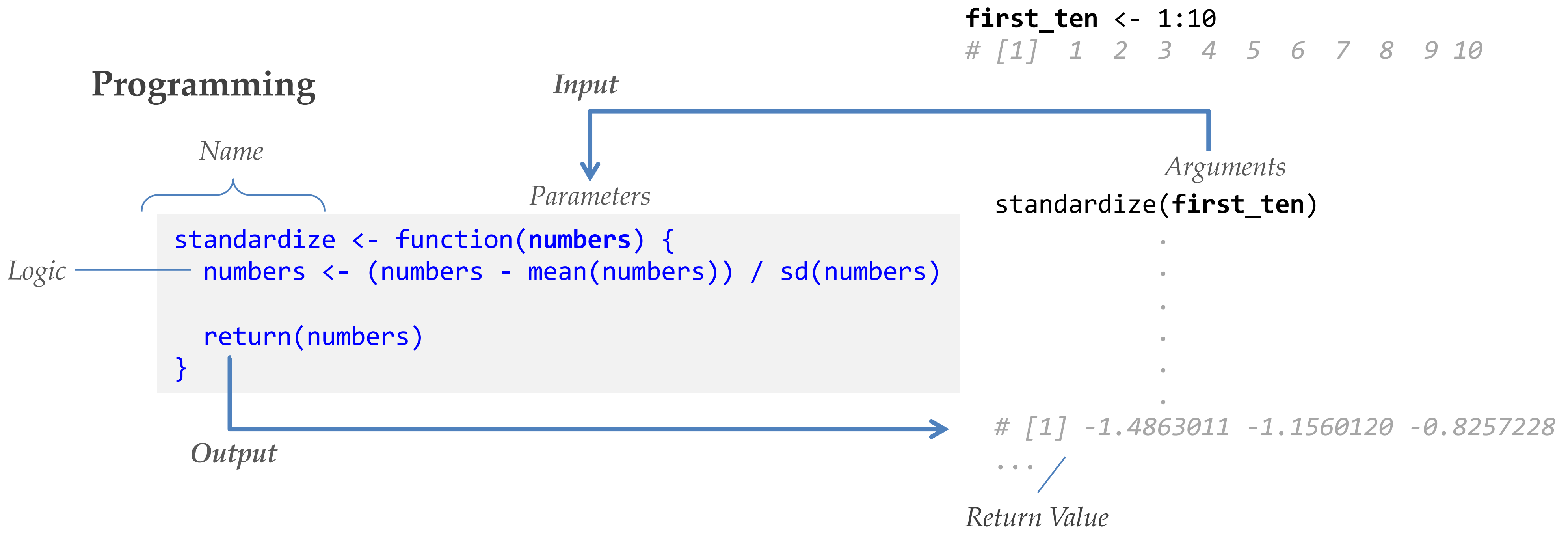
Functions

Math

Functions define how certain variable y are associated with with other variables x

$$y = f(x)$$

Programming



```r
first_ten <- 1:10
# [1]  1  2  3  4  5  6  7  8  9 10
```

```r
standardize <- function(numbers) {
  numbers <- (numbers - mean(numbers)) / sd(numbers)

  return(numbers)
}
```

```r
standardize(first_ten)
# [1] -1.4863011 -1.1560120 -0.8257228
...
```

Look carefully…

```r
numbers <- (numbers - mean(numbers)) / sd(numbers)
```

Is the input vector `number` is changed in function?

🤔 *Will `standardize()` change the `first_ten` vector?*

Changing the value of variables

```
standardize <- function(numbers) {
  numbers <- (numbers - mean(numbers)) / sd(numbers)

  return(numbers)
}
```

Look carefully…

```
numbers <- (numbers - mean(numbers)) / sd(numbers)
```

Is the input vector `number` *is changed in function?*

🤔 *Will* `standardize()` *change the* `first_ten` *vector?*

Can a function change an input?

```
first_ten <- 1:10
# [1]  1  2  3  4  5  6  7  8  9 10

standardized(first_ten)
# [1] -1.4863011 -1.1560120 -0.8257228 ...

first_ten
# [1]  1  2  3  4  5  6  7  8  9 10
```

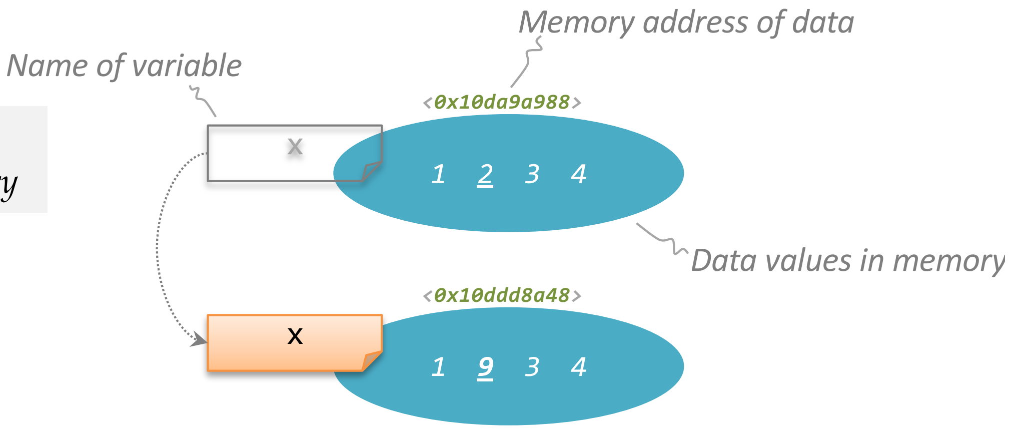
It seems the `first_ten` *vector was* ***not changed***

🤔 *What happened when* `numbers` *was changed inside the function?*

Let's see what happens when the value of variables are changed in R:

```
x <- c(1, 2, 3, 4)

tracemem(x)
[1] "<0x10da9a988>"

x[2] <- 9
tracemem[0x10da9a988 -> 0x10ddd8a48]:

x
[1] 1 9 3 4
```

*`tracemem()` shows us the **address** where a variable's data is stored in computer memory*



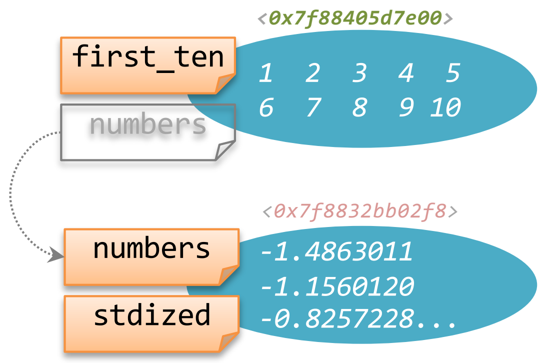
Let's see what happens to variables changed inside functions:

```
standardized_mem <- function(numbers) {
  cat(paste("BEFORE change: numbers is at: ", tracemem(numbers), "\n"))
  numbers <- (numbers - mean(numbers)) / sd(numbers)
  cat(paste("AFTER change:  numbers is at: ", tracemem(numbers), "\n"))
  return(numbers)
}
```

R does
"copy on change"
(makes a new copy of data if it is changed)

```
tracemem(first_ten)
# [1] "<0x7f88405d7e00>"

stdized <- standardized_mem(first_ten)
# BEFORE change: numbers is at:  <0x7f88405d7e00>
# AFTER change:  numbers is at:  <0x7f8832bb02f8>

tracemem(stdized)
# [1] "<0x7f8843b6d318>"
```

*R **passes objects** to functions **by reference***



For Loops in R

A procedural way to repeat operations in code

$\underbrace{\text{procedural}}_{\text{Telling the computer what to do, rather than what you want}}$

Problem: Given a data vector,

Create a vector of boolean (true/false) values to indicate if each data element is positive or not

```
norm_data <- rnorm(500000)
# [1] 0.01874617 -0.18425254 -1.37133055 -0.59916772  0.29454513  0.38979430…


# [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE…
```

One Solution: Create a loop that checks each element in the vector of data and makes a new vector of (-1) or (+1) values

*Setup **shadow variable***

*Perform **logic***

*Build **results** value to return*

```
results <- c()

for (i in 1:length(norm_data)) {
  if (norm_data[i] < 0) {
    results[i] <- FALSE
  } else {
    results[i] <- TRUE
  }
}

# [1]  TRUE  FALSE  FALSE  FALSE   TRUE   TRUE…
```

*Define **index** range to loop over*

- *"i" takes next value in sequence in each iteration*

The Problem with For-loops

for-loop with *index*

```
results <- c()
for (i in 1:length(norm_data)) {
  if (norm_data[i] < 0) {
    results[i] <- FALSE
  } else {
    results[i] <- TRUE
  }
}

# [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE…
```

Shadow variables *make code harder to understand*

Hard to know the intention of the loop
Made for computers to understand
Hard for people to understand

Easy to write slow, unoptimal code
Many optimization tricks must be learned
Harder to perform in parallel

for-loop without *index*

```
results <- c()
for (num in norm_data) {
  if (num < 0) {
    results <- c(results, FALSE)
  } else {
    results <- c(results, TRUE)
  }
}
```

🤔

Which version of the for-loop is easier to understand?
(try reading each!)

Which version of the for loop will be faster? Why?
(try running each!)

Advantages of for-loops?
- *More familiar to programmers from older languages*
- *Helpful if index of operations is important*

Functional Iteration

Stating your purpose (function) rather than procedure

Apply family (apply/sapply/lapply)

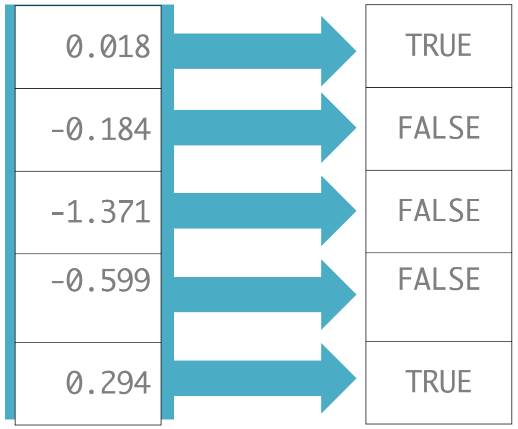
1. Define a iteration logic in a **function**

```
is_positive <- function(num) {
  if (num > 0) {
    return(TRUE)
  } else {
    return(FALSE)
  }
}

is_positive(5)
[1] TRUE
```

Intention is clearer than for-loops
We wish to apply the logic of `positive()` *function to every element of norm_data*

2. **Apply** the function over the data

```
results <- sapply(norm_data, is_positive)
# [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE
```



Doesn't have to be optimized like a for-loop
R can optimize your functional iterations
Can be parallelized (see ***`parallel`*** *package)*
But still not always the fastest or clearest way…

R treats functions as first-class objects:
functions can be stored like data
functions can be passed as arguments
functions can even return functions

E.g., our function is passed to `sapply()` *as an* *argument*

```
sapply(norm_data, is_positive)
```

Vectorized Iteration

R often already loops through data that is in vector form!

Most readable

vector output *vector input* *greater-than operator*

*Most **operators** are already vectorized*

```
results <- norm_data > 0
# [1]  TRUE FALSE FALSE FALSE  TRUE  TRUE…
```

*Some **functions** are vectorized as well*

```
ifelse(norm_data > 0, "positive", "negative")
# [1] "positive" "negative" "negative" "negative"
```

Matches mathematical representation

Vectorized code is most most likely to resemble your math/statistics

$$x - \bar{x}$$

```
x - mean(x)
```

$$\frac{\sum(x - \bar{x})^2}{n - 1}$$

```
sum((x - mean(x))^2) / (length(x) - 1)
```

Benchmarking Performance

`system.time(...)`

If performance is critical, you can ***benchmark*** *the time performance of your code*

For loops *can be very fast, or very slow, depending on your experience and the code*

```
results <- c();
system.time(
  for (i in 1:length(norm_data)) {
    if (norm_data[i] < 0) { results[i] <- FALSE}
    else { results[i] <- TRUE }
  }
)
#  user   system elapsed
# 0.159    0.020    0.186
```

elapsed time is the total time

Functional iteration *performs somewhere between fast and slow for-loops*

```
system.time( sapply(norm_data, positive) )
#  user   system elapsed
# 0.329    0.010    0.340
```

Vectorized iteration *is always fastest, if it is available*

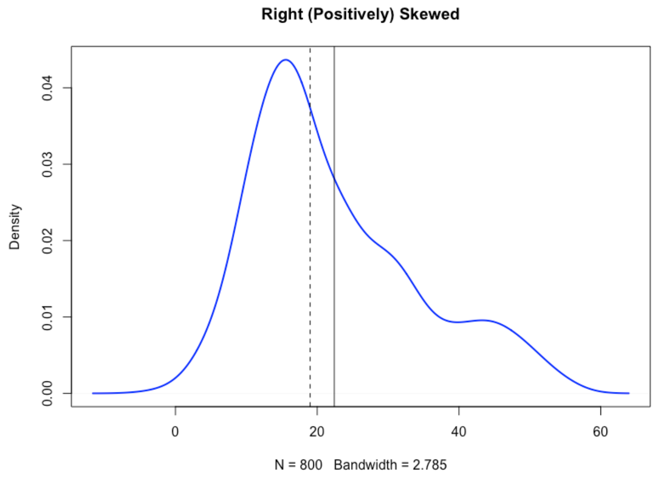
```
system.time( norm_data > 0 )
user   system elapsed
0.001   0.000    0.001
```

Making your own function *can help make repetitive tasks quicker and easy to change*

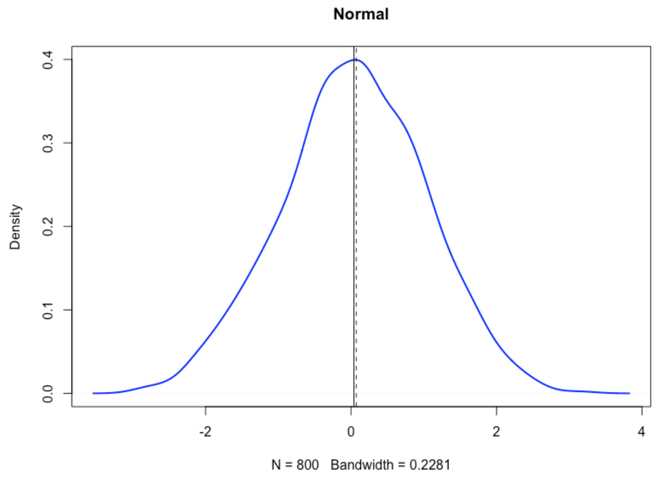
```
# Create a visualization function
plot_centrality <- function(distr, title) {
  # Plot the full distribution of abc
  plot(density(distr), col="blue", lwd=2, main = title)

  # Add vertical lines showing mean and median
  abline(v=mean(distr))
  abline(v=median(distr), lty="dashed")
}
```

Mean vs. Median

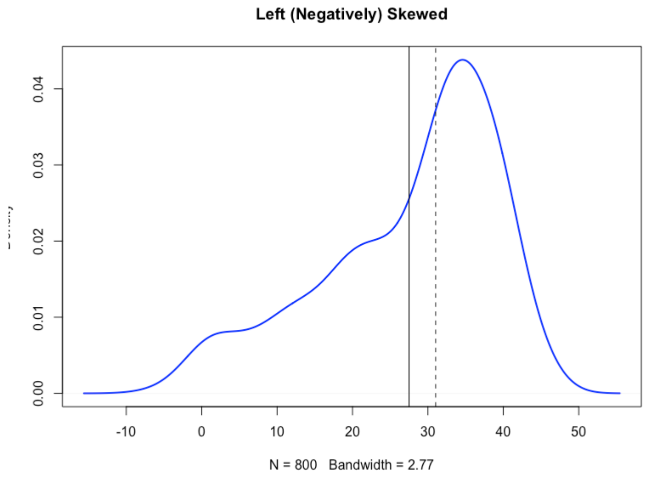
The ***mean appears to be more sensitive*** *to outliers*
The median (and other quartiles) appears to be more resilient to outliers



```
d1 <- rnorm(n=500, mean=15, sd=5)
d2 <- rnorm(n=200, mean=30, sd=5)
d3 <- rnorm(n=100, mean=45, sd=5)
d123 <- c(d1, d2, d3)
plot_centrality(
  d123,
  title="Right (Positively) Skewed”
)
```



```
d123 <- rnorm(n=800)
plot_centrality(d123, title="Normal")
```



```
d1 <- rnorm(n=100, mean=5, sd=5)
d2 <- rnorm(n=200, mean=20, sd=5)
d3 <- rnorm(n=500, mean=35, sd=5)
d123 <- c(d1, d2, d3)
plot_centrality(
  d123,
  title="Left (Negatively) Skewed”
)
```

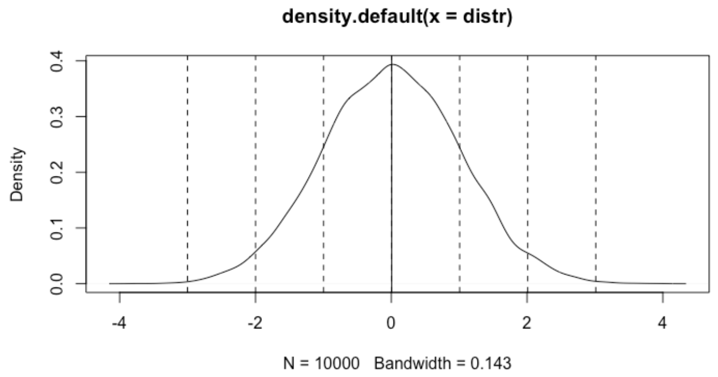
🤔

Why do we always use the ***mean*** *in statistics?*

Could we do statistics with the ***median****?*

Standard Deviations of a Normal Distribution

```
quartiles_vs_sd <- function(distr) {
  # plot data distribution, mean + standard deviations lines
  plot(density(distr))
  abline(v=mean(distr))
  sd_points <- mean(distr) + (-3:3)*sd(distr)
  abline(v=sd_points, lty='dashed')

  # return the distance of each quartile from the mean
  q = quantile(distr, c(0.25, 0.50, 0.75))
  return((q - mean(distr))/sd(distr))
}
```



Standard Normal Distribution *(mean=0, sd=1)*?

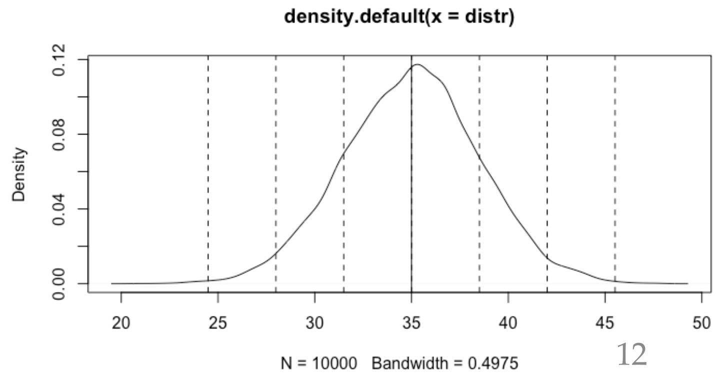
```
data = rnorm(n=10000, mean=0, sd=1)
quartiles_vs_sd(data)
         25%          50%          75%
-0.669003605 -0.005018409  0.678760278
```

*Our quartile deviations from the mean are similar if expressed in terms of **standard deviations***

*Our quartile deviations from the mean are quite different in terms of their **raw units***

Non-standard Normal Distribution? *(sd≠1)*

```
data = rnorm(n=10000, mean=35, sd=3.5)
quartiles_vs_sd(data)
         25%          50%          75%
-0.673143918  0.001196753  0.672708091
```



Quartiles and Standard deviation:
- *Describes dispersion in normal data, regardless of its center*
- *Can give us a standard sense of range for normal distributions*

The Empirical Rule

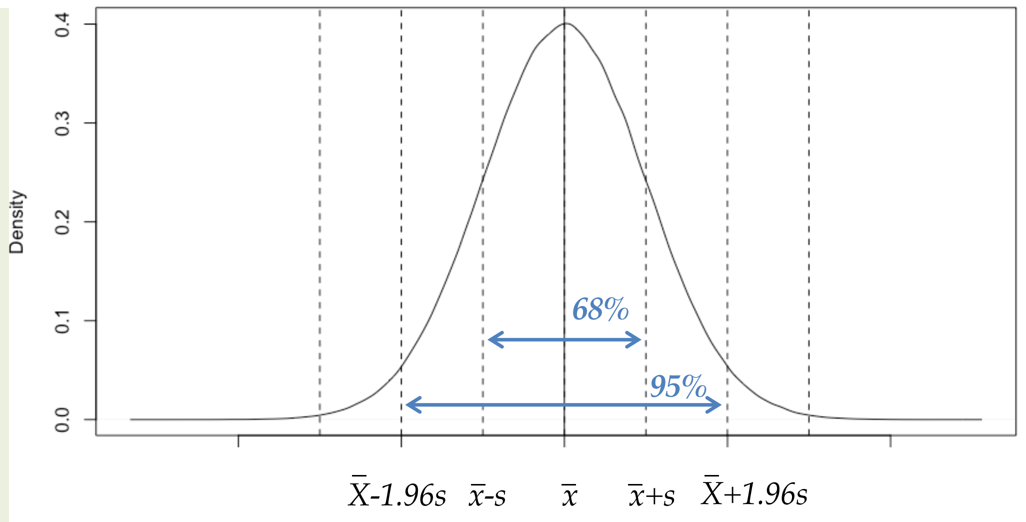
For any symmetrical, bell-shaped distribution

68% of cases are within 1 sd of the mean

```
mean(data) + c(-1, 1)*sd(data)
[1] 31.49610 38.50571
```

95% of cases are within 1.96 sd of the mean

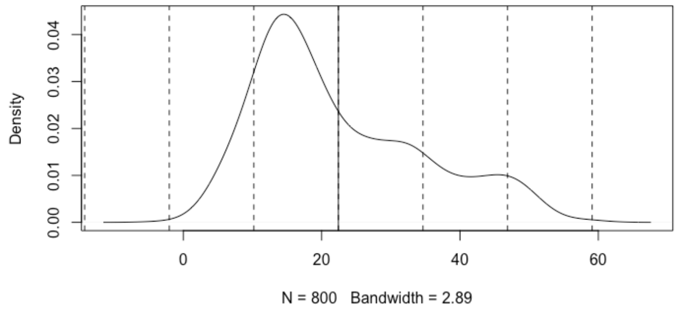
```
mean(data) + c(-1.96, 1.96)*sd(data)
[1] 28.13150 41.87031
```



28.13 31.50 38.51 41.87

Does not apply for other distributions

```
> quartiles_vs_sd(d123)
        25%        50%        75%
-0.7513032 -0.3136625  0.6942677
```



Standard deviation does not describe familiar points in non-normal data

Are ***quartiles*** *a better description of dispersion in non-normal data?*

Interquartile Range Revisited

🤔 *Should we prefer these metrics to make plots, or should we rely on our own expertise ?*

normal data | **normal + outliers**

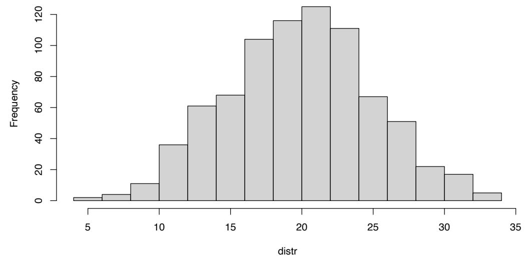
Sturges' Rule: Log length for number of bins

```r
sturges_formula <- function(distr, title) {
  k = ceiling(log2(length(distr))) + 1
  h = (max(distr) - min(distr))/k
  hist(distr, breaks = k)
  return(data.frame(k, h))
}
```

Log length keeps number of bins (k) stable, which is not good when outliers are present, but might be good for other things?

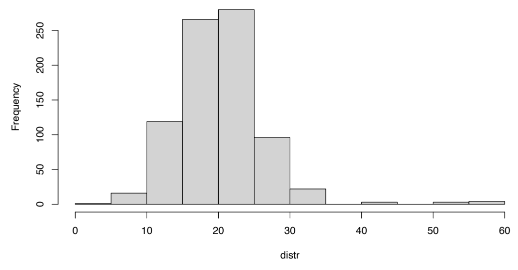
```r
sturges_formula(rand_data)
```

```
  k          h
 11       2.60
```



```r
sturges_formula(out_data)
```

```
  k          h
 11       4.99
```



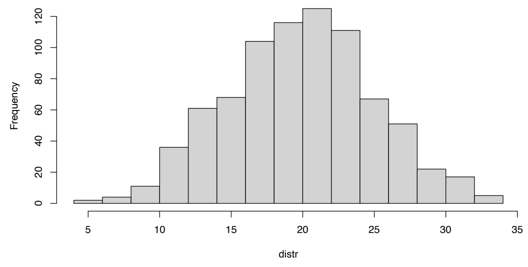
Scott's Rule: Standard deviation for bin size

```r
scotts_rule <- function(distr) {
  h <- 3.5*sd(distr) / (length(distr)^(1/3))
  k = ceiling((max(distr) - min(distr))/h)
  hist(distr, breaks = k)
  return(data.frame(k, h))
}
```

Standard deviation is stabler, but could cause problems if extreme outliers are introduced

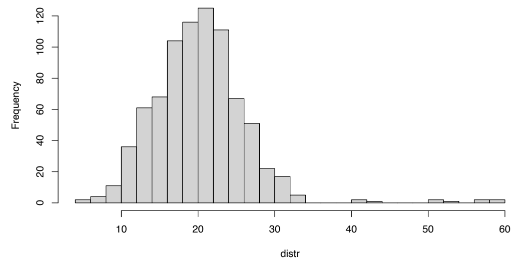
```r
scotts_rule(rand_data)
```

```
  k          h
 15       1.92
```



```r
scotts_rule(out_data)
```

```
  k          h
 24       2.32
```



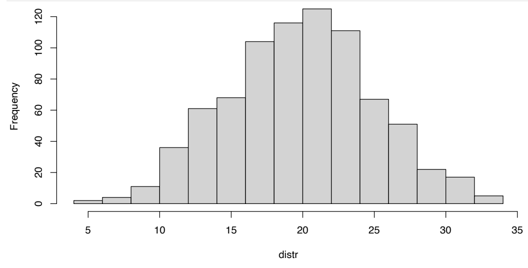
Freedman-Diaconis Choice: IQR for bin size

```r
fd_choice <- function(distr, title) {
  h <- 2*IQR(distr) / (length(distr)^(1/3))
  k <- ceiling((max(distr) - min(distr))/h)
  hist(distr, breaks = k, main=title)
  return(data.frame(k, h))
}
```

Using quartiles (IQR) makes bin size (h) insensitive to the presence of outliers

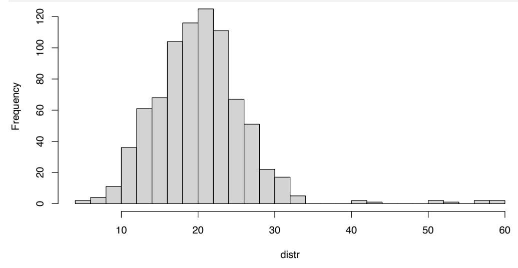
```r
fd_choice(norm_data)
```

```
  k        h
 20     1.49
```



```r
fd_choice(rand_data)
```

```
  k        h
 38     1.48
```



HW Peer Review Metric

	Completeness		Solution		Extra
5	Everything attempted	*and*	Everything correct	*and*	Novel solution or Beautiful report
100% = **4**	**Everything attempted**	***and***	**Everything correct**	***and***	**Reasonable appearance**
3	Everything attempted	*and*	Minor mistakes	*or*	Difficult to interpret
2	**Missing** major parts or **late submission**	*or*	**Major mistakes**		
1					
0	Not submitted				

HW Suggestions

CREATE well formatted reports

Briefly summarize the question

Format it to distinguish:
question / description / code / output / answers

Show code and relevant text output
use text, not screenshots

Show relevant visualizations
export graphics from Rstudio; not screenshots

CREDIT peers who helped

Mention their ID at the top of your assignment
Peers who help will get extra-credit at end-of-semester

REVIEW your peers fairly

Give specific comments for your response (0-5)
Why the homework was deducted any points
Why the homework was awarded 5 points

You will get a 'reviewer' grade at end-of-semester:
accuracy, helpfulness

If you aren't happy with your peer evaluation

Politely reply to your peer evaluations with a comment
we will check comments!

Contact the TAs or professor if no response

Resampling from a Population

The Central Limit Theorem

```
library(compstatslib)
d3 <- rnorm(n=500000, mean=5, sd=5)
d2 <- rnorm(n=200000, mean=20, sd=5)
d1 <- rnorm(n=100000, mean=35, sd=5)
d123 <- c(d1, d2, d3)
interactive_sampling(d123)
```

distribution of population :
(distribution unknown)

Population Mean: μ_x

Standard Deviation: σ_x

distribution of a sample:

Sample Mean:
(weakly approx. to pop. mean)

$$\bar{x} = \frac{\sum x_i}{n} \sim \mu_x$$

Standard Deviation:

$$s = \sqrt{\frac{\sum(x_i - \bar{x})^2}{n-1}}$$

distribution of all sample means:

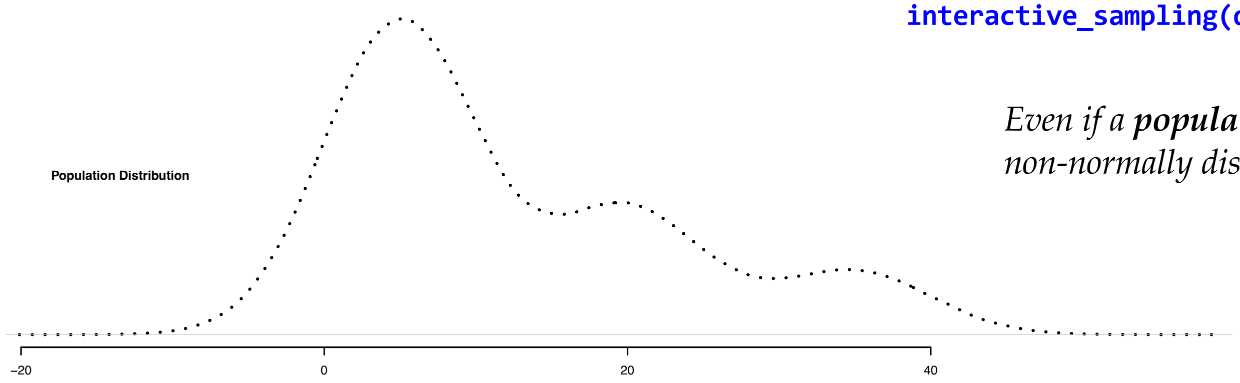
$$\bar{x}_1, \bar{x}_2, \bar{x}_3, \ldots, \bar{x}_{100}$$

Follows a nearly normal distribution

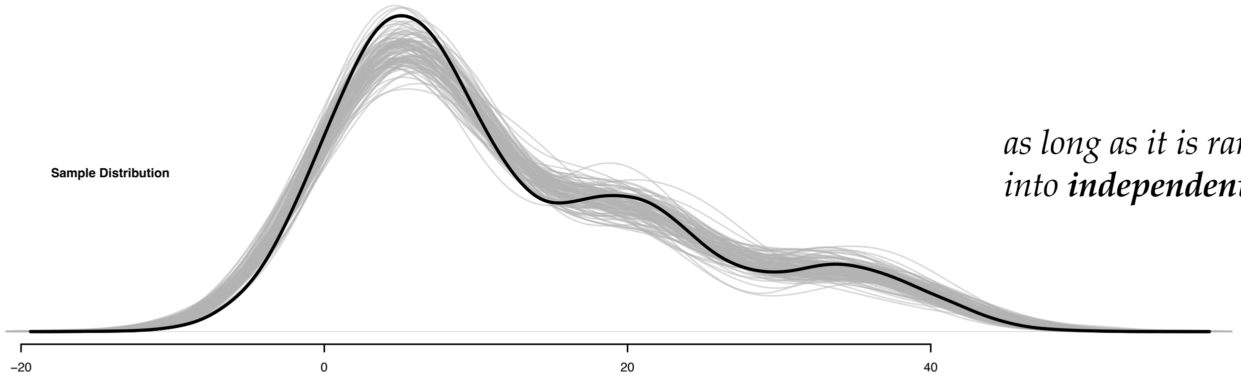
We can approximate its mean and standard deviation with a single sample:

$$\bar{\bar{x}} \sim \bar{x}$$

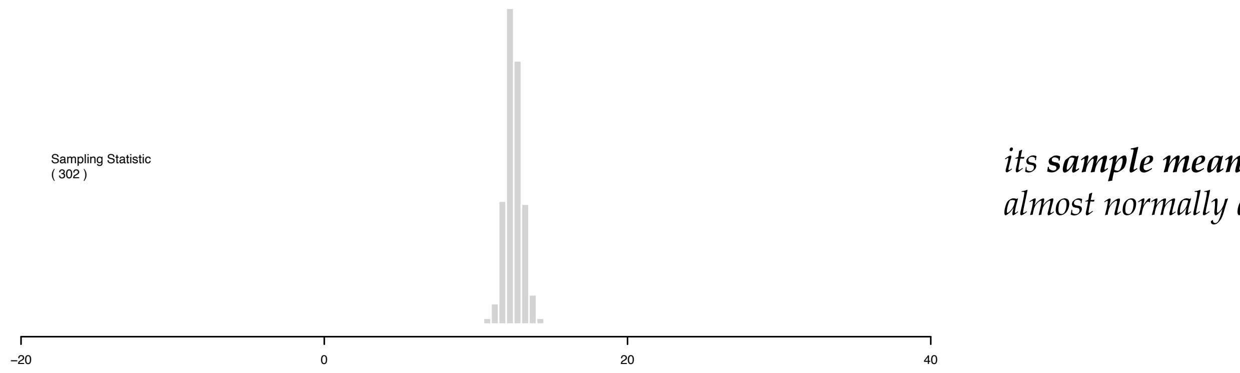
$$s_{\bar{x}} = \frac{s_x}{\sqrt{n}}$$



*Even if a **population** is non-normally distributed,*



*as long as it is randomly drawn into **independent samples**,*



*its **sample means** will be almost normally distributed.*

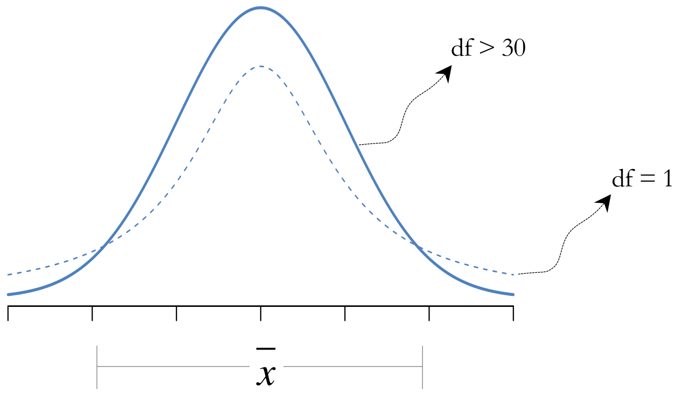
Confidence Interval of μ

```
https://gist.github.com/soumyaray/285296600b8712b04b52201010bbbd9f
```

Population statistics:

Distribution characteristics unknown

distribution of sample means $\bar{x}_1, \bar{x}_2, \bar{x}_3, \ldots, \bar{x}_{100}$



Sample statistics:

Sample Mean:
(weakly approx. to pop. mean)

$$\bar{x} = \frac{\sum x_i}{n} \sim \mu_x$$

Standard Deviation:

$$s = \sqrt{\frac{\sum (x_i - \bar{x})^2}{n-1}}$$

sample size: n

degrees of freedom **(df)** = *n*-1

Standard error:
(based on one sample)

$$s_{\bar{x}} = {}^{s}/_{\sqrt{n}}$$

The population mean should somewhere in the distribution of sampling means

95% Confidence Interval of the mean: $\bar{x} - 1.96\left({}^{s}/_{\sqrt{n}}\right)$ *to* $\bar{x} + 1.96\left({}^{s}/_{\sqrt{n}}\right)$

99% Confidence Interval of the mean: $\bar{x} - 2.58\left({}^{s}/_{\sqrt{n}}\right)$ *to* $\bar{x} + 2.58\left({}^{s}/_{\sqrt{n}}\right)$

Confidence Interval *of Population Mean (μ_x):*

$$\bar{x} \pm t\left({}^{s}/_{\sqrt{n}}\right)$$

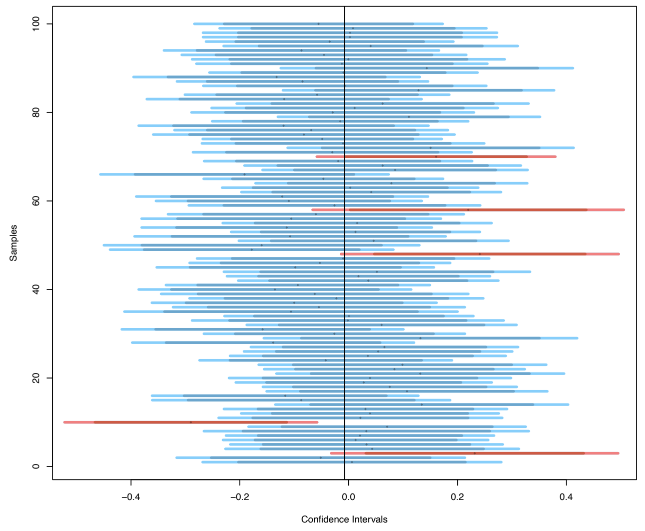
Confidence Level	t (*df* > 30)
90.0%	**1.65**
95.0%	**1.96**
99.0%	**2.58**

If we take a large number of samples,

~95% of samples should contain the population mean in their 95% confidence interval,

~99% of samples should contain the population mean in their 99% confidence interval

```
library(compstatslib)
plot_sample_ci()
```



Resampling with Replacement

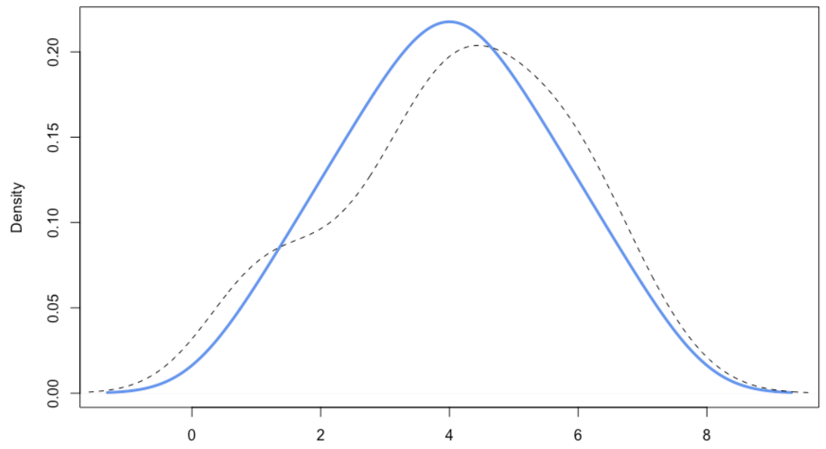
Resampling from a Uniform Distribution:

```
seq_sample <- 1:22
resample <- sample(seq_sample, replace=TRUE)

seq_sample
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22
sort(resample)
[1]  1  1  3  4  6  8  9  9 13 13 13 14 14 15 15 16 17 20 20 20 20 22
```

Recall: we learned how to "sample with replacement"
Is this data simulation helpful?

Resampling from a Bell-shaped Distribution

```
rep_sample <- c(1, rep(2,2), rep(3,3), rep(4,4), rep(5,3), rep(6,2), 7)
resample <- sample(rep_sample, replace=TRUE)

rep_sample
[1] 1 2 2 3 3 3 4 4 4 4 5 5 5 6 6 7

sort(resample)
[1] 1 1 2 3 3 4 4 4 4 5 5 5 6 6 6 7


plot(density(rep_sample), lwd=3)
lines(density(resample), lty="dashed")
```



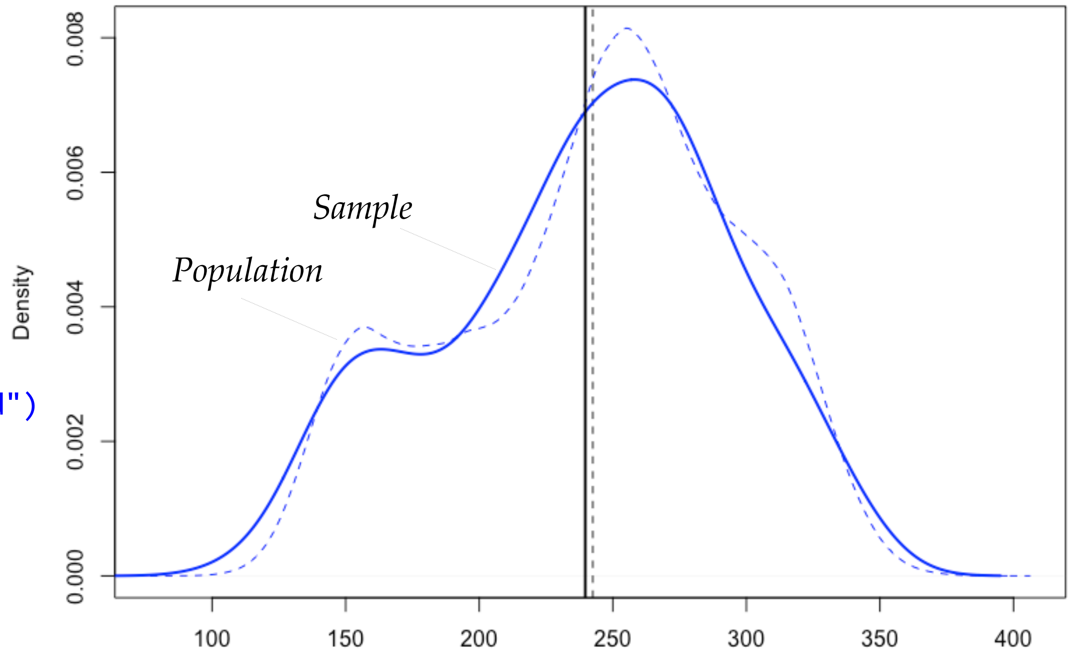
If we randomly pick elements from a sample with replacement,
our new sample will have a ***similar distribution*** *to our original sample*

Population vs. Sample

The Unseen Population

Imagine a population that we cannot measure

```
a <- rnorm(n=100000, mean=150, sd=15)
b <- rnorm(n=200000, mean=190, sd=25)
c <- rnorm(n=500000, mean=255, sd=25)
d <- rnorm(n=200000, mean=310, sd=20)

population <- c(a,b,c,d)
pop_mean <- mean(population)

plot(density(population), col="blue", lty="dashed")
abline(v=pop_mean, lty="dashed")

pop_mean
# [1] 242.5034
```



Taking One Sample

Let's take a sample (`sample0`) from the population: it's the only thing we can measure

```
sample_size = 300
sample0 = sample(pop, sample_size)

sample0_mean = mean(sample0)
```

The sample will not have exactly the same distribution or descriptives as the population

```
lines(density(sample0), col="blue", lwd=2)
abline(v=sample0_mean, lwd=2)

sample0_mean
# [1] 239.6902
```

🤔

But resampling loses much of the information about the population

Can we say anything about the population?

The **Bootstrap**: Computational Resampling for Inference

$\overbrace{\text{Bootstrap}}$

To start a process with no input or help
"bootup the computer"
"pick yourself up by your bootstraps!"

We can ***resample*** *from our* ***original sample*** *to see where most of the sampling means are.*

Bootstrapped resamples

```
resamples <- replicate(3000,
                       sample(sample0, length(sample0), replace=TRUE))

dim(resamples)
[1]  300 3000
```

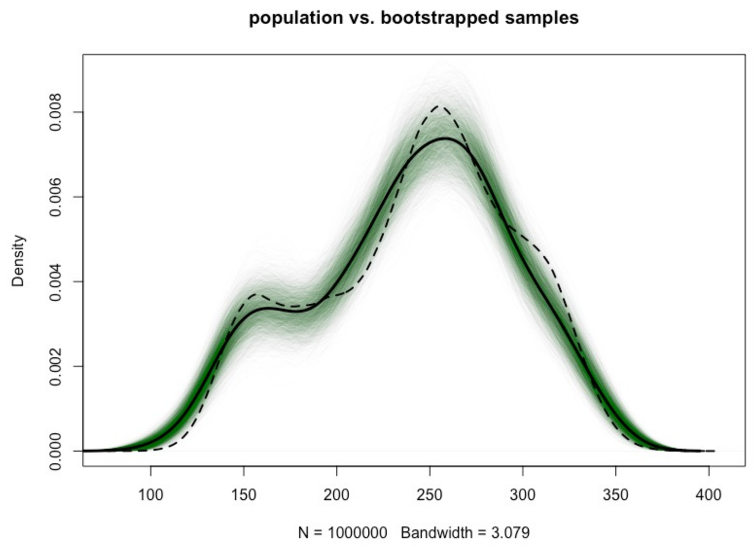
`replicate` *returns* ***300 rows*** *(data in each sample) and* ***3000 columns*** *(samples)*

`replicate(n, expr)`:
repeat an operation n times

Visualizing our resampled samples

```
# Create an empty plotting space with axes
plot(density(population), lwd=0, ylim=c(0, 0.009))

# A function to plot a single sample's distribution
plot_resample_density <- function(sample_i) {
  lines(density(sample_i), col=rgb(0.0, 0.4, 0.0, 0.01))
  return(mean(sample_i))
}

# Iteratively plot and get means of all bootstrapped samples
sample_means <- apply(resamples, 2, FUN=plot_resample_density)

# Plot hidden population and original sample distributions
lines(density(sample0), lwd=3)
lines(density(population), lwd=2, lty="dashed")
```

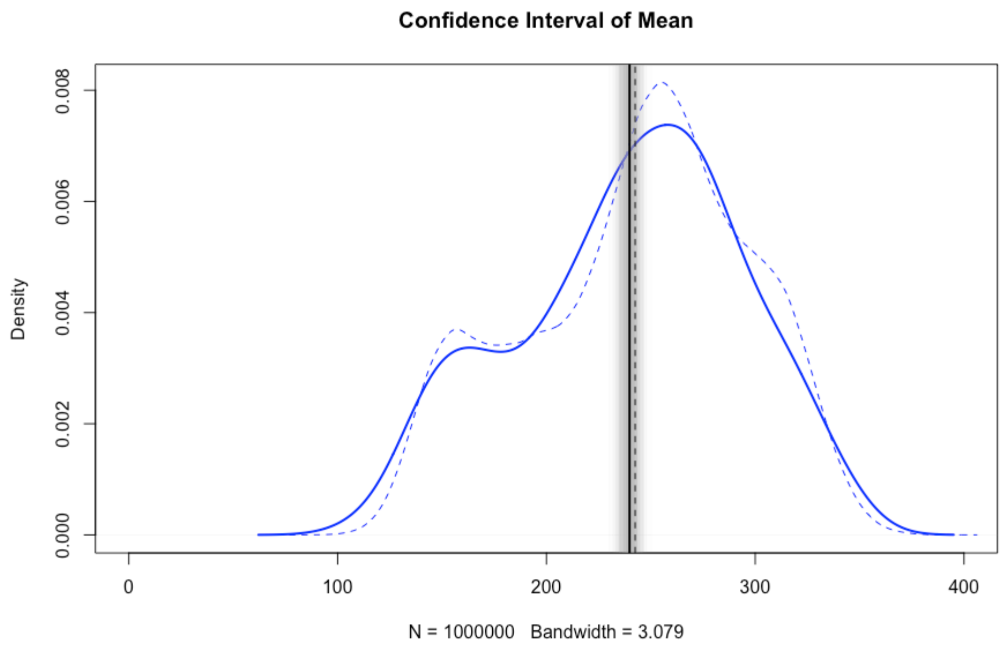
`plot_resample_density()` *will plot and compute mean for a single resample*



```
rgb(0.0, 0.4, 0.0, 0.01)
     red  green blue  alpha
                      transparency
```

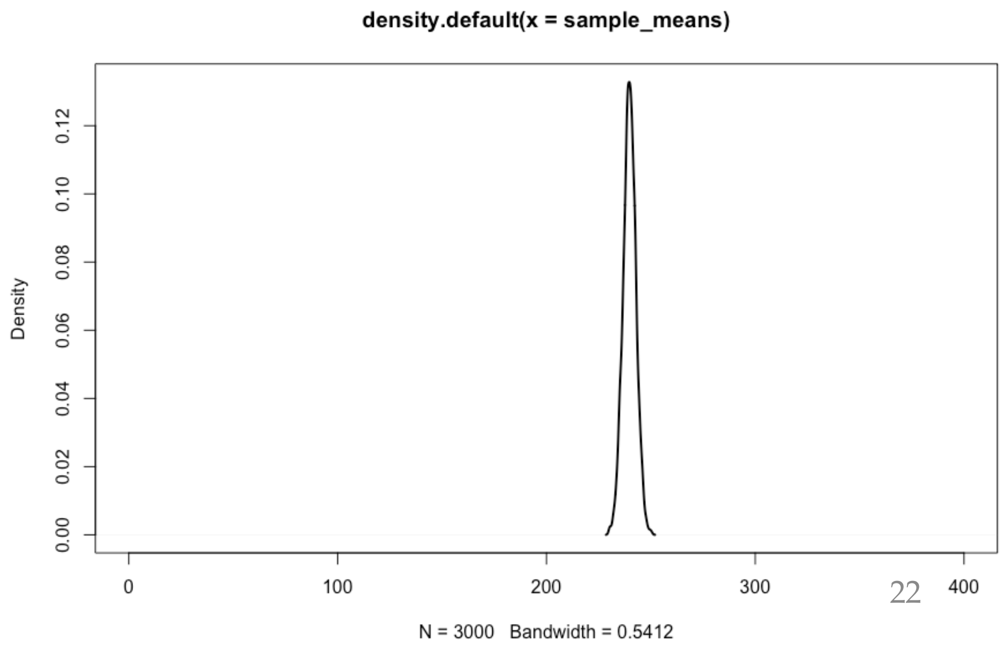
Visualizing Resampled Means

```
# Plot population and original sample densities
plot(density(pop), col="blue", lty="dashed")
lines(density(sample0), col="blue", lwd=2)

# Draw light vertical lines for each sampling mean
abline(v=sample_means, col=rgb(0.7, 0.7, 0.7, 0.01))

# Draw dark lines of population and original sample mean
abline(v=mean(sample_means), lwd=2)
abline(v=pop_mean, lty="dashed")
```



Distribution of Resampled Means

```
## Distribution of sampling
plot(density(sample_means), lwd=2, xlim=c(0, 400))

## Confidence intervals of the sampling means
quantile(sample_means, probs=c(0.025, 0.975))
#     2.5%    97.5%
# 233.7158 245.9080

quantile(sample_means, probs=c(0.005, 0.995))
#     0.5%    99.5%
# 231.8270 247.8224
```


density.default(x = sample_means)
Density
N = 3000   Bandwidth = 0.5412