Control Structures

Adam Kuczynski

UNIVERSITY OF WASHINGTON

Control structures allow you to control the flow of your R program (script) and are critical to programming in R

There are three components of a program's flow:

1. **Sequential**: the order in which the R code is executed

   > "do *this* first, then *that*"

2. **Selection**: which path of an algorithm R will execute based on certain criteria

   > "do *that*, but only if *this* is `TRUE`

   - `if`/`else`
   - `switch`

3. **Iteration**: how many types should a certain algorithm be repeated?

   > "do *this* 100 times, then move on to *that*"

   - `for`, `while`, `repeat`
   - `break`, `next`

`if()` otherwise `else`

`if()` and `else` statements allow you to conditionally execute code

For example, write a program that tells a cashier whether or not they should sell alcohol to a customer. The cashier enters the customer's birthday into their POS, which needs to display the appropriate message:

```r
# Calculate age
age <- as.numeric(difftime(Sys.time(), as.Date(birthday))) / 365

if(age >= 21){
  print(paste("Age:", floor(age), "(Sell)"))
} else {
  print(paste("Age:", floor(age), "(Do not sell)"))
}
```

```r
birthday <- "2002-05-20"
```

```
## [1] "Age: 19 (Do not sell)"
```

```r
birthday <- "1970-12-15"
```

```
## [1] "Age: 50 (Sell)"
```

`if()` statements do not need an else statement, if there is no alternative

```r
number <- 31

# If number is even
if(number %% 2 == 0){
  print("Congratulations! It's an even number.")
}
```

You can also have multiple `if/else` statements in a row, if there are more than two outcomes

```r
number <- 27

if(number %% 2 == 0){
  print("Congratulations! It's an even number!")
} else if(number %% 3 == 0){
  print("Your number is divisible by 3")
} else {
  print("Your number is not even or divisible by 3")
}
```

```
## [1] "Your number is divisible by 3"
```

The statements evaluated by `if()` *always* need to return a single `TRUE` or `FALSE` value

```r
if(TRUE){
  print("This will always run")
} else {
  print("This will **never** run")
}
```

```
## [1] "This will always run"
```

```r
if("Character"){
  print("Take the course")
}
```

```
## Error in if ("Character") {: argument is not interpretable as logical
```

Be careful, though! R will coerce values inside `if()` in unexpected ways:

```r
if(359){
  print("This code ran")
}
```

```
## [1] "This code ran"
```

```r
if("TRUE"){
  print("This code ran")
}
```

```
## [1] "This code ran"
```

You can have multiple conditions inside an `if()` statement as well with `else if()`

```r
age <- 36
hasmoney <- TRUE

if(age > 21 && hasmoney){
  print("Sell!")
} else if(age > 21 && !hasmoney){ # of age, but no money
  print("Sell when they have money!")
} else {
  print("Do not sell")
}
```

```
## [1] "Sell!"
```

```r
age <- 36
hasmoney <- FALSE
```

```
## [1] "Sell when they have money!"
```

```r
age <- 12
hasmoney <- TRUE
```

```
## [1] "Do not sell"
```

`ifelse()` and `if_else()`

`if/else` takes *one* `TRUE` or `FALSE` value, but sometimes we want to evaluate multiple values at once

`ifelse()` is a **vectorized** version of `if/else` that can operator over vectors:

```r
ages <- c(35, 12, 82, 21, 15)
ifelse(ages > 21, "Sell alcohol", "Do not sell alcohol")
```

```
## [1] "Sell alcohol"        "Do not sell alcohol" "Sell alcohol"
## [4] "Do not sell alcohol" "Do not sell alcohol"
```

`ifelse()` if very useful inside a dataframe to transform your data.

Remember the `uwclinspsych` dataframe from last week? 👉

```
##     name grads fullprof
## 1  Corey     1    FALSE
## 2 Angela     0    FALSE
## 3   Bill     4     TRUE
## 4   Mary     3     TRUE
## 5   Jane     2     TRUE
## 6   Lori     3     TRUE
```

UNIVERSITY OF WASHINGTON

Lets create a new variable inside `uwclinpsych` called `newgrad` that is the number of grad students each faculty is allowed to take this year

If a faculty has 3+ grad students they aren't allowed to take any, but if they have 0-2 they are allowed to take up to 2:

```
uwclinpsych$newgrad <- ifelse(test = uwclinpsych$grads >= 3,
                              yes = 0,
                              no = 2)
print(uwclinpsych)
```

```
##     name grads fullprof newgrad
## 1  Corey     1    FALSE       2
## 2 Angela     0    FALSE       2
## 3   Bill     4     TRUE       0
## 4   Mary     3     TRUE       0
## 5   Jane     2     TRUE       2
## 6   Lori     3     TRUE       0
```

`if_else()` from the `dplyr` package[1] is *very* similar to base R's `ifelse()` except it makes sure the return values are the same type:

```r
mylets <- factor(sample(letters[1:5], 10, replace = TRUE))
print(mylets)
```

```
##  [1] b d d e a d e b e a
## Levels: a b d e
```

```r
ifelse(mylets %in% c("a", "b", "c"), mylets, factor(NA))
```

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

```r
dplyr::if_else(mylets %in% c("a", "b", "c"), mylets, factor(NA))
```

```
##  [1] b    <NA> <NA> <NA> a    <NA> <NA> b    <NA> a
## Levels: a b d e
```

[1] The `dplyr` package is written by Hadley Wickham and is part of the Tidyverse.

`switch()`

`switch()` operates in much the same way as `if/else` statements by letting you select among a list of alternatives given one input value

`switch()` is useful when you have:

- one single test condition
- your test condition is character or an integer representing an index within a list of option
- you have 2+ conditions

From `help(switch)`:

```r
switch(EXPR, ...)
```

- `EXPR` = an expression evaluating to a number or a character string
- ... the list of alternatives. If it is intended that EXPR has a character-string value these will be named, perhaps except for one alternative to be used as a ‘default’ value.

If `EXPR` is **numeric**, R will return the list of alternatives corresponding with that index:

```r
switch(1,
       "First",
       "Second",
       "Third")
```

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

```r
switch(3,
       "First",
       "Second",
       "Third")
```

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

If `EXPR` is **character**, R will search the list of alternatives and return the associated value:

```r
switch("this",
       this = "This one!",
       that = "That one!",
       "The other one!")
```

```
## [1] "This one!"
```

```r
switch("Not this or that!",
       this = "This one!",
       that = "That one!",
       "The other one!")
```

```
## [1] "The other one!"
```

☝ Notice the unnamed argument at the end! This is optional. ☝

Using `switch()` prevents you from having to write a bunch of `if/else` statements (it can also result in faster code, but this is generally negligible):

```r
switch("this",
       this = "This one!",
       that = "That one!",
       "The other one!")
```

☝️ is equivalent to 👇

```r
if(x == "this"){
  "This one!"
} else if(x == "that"){
  "That one!"
} else {
  "The other one!"
}
```

Loops

Computers are *really* good at repeating the same task over and over, and loops are the way to accomplish it

From Wikipedia:

> A loop is a sequence of statements which is specified once but which may be carried out several times in succession. The code "inside" the loop is obeyed a specified number of times, or once for each of a collection of items, or until some condition is met, or indefinitely."

There are three types of loops in R:

- `for` loops
- `while` loops
- `repeat` loops

Bad repetition: Let's say you wanted to take the mean of all columns in the `swiss` dataset:

```
mean1 <- mean(swiss$Fertility)
mean2 <- mean(swiss$Agriculture)
mean3 <- mean(swissExamination)
mean4 <- mean(swiss$Fertility)
mean5 <- mean(swiss$Catholic)
mean5 <- mean(swiss$Infant.Mortality)
c(mean1, mean2 mean3, mean4, mean5, man6)
```

Can you spot the problems with this code?

How frustrated would you be if `swiss` had 200 columns instead of 6?

DRY vs. WET Programming

DRY: do not **r**epeat **y**ourself! If you are wriing the the same code over several lines, there's probably a more efficient way to write it

WET:

- **w**rite **e**very **t**ime
- **w**rite **e**verything **t**wice
- **w**e **e**njoy **t**yping
- **w**aste **e**veryone's **t**ime

Writing DRY code reduces risk of making typos in your code, *substantially* reduces the time and effort involves in processing large volumes of data, and is more readable and easier to troubleshoot

`for` Loop

`for` loops iterate over a vector of values (any atomic type!) and execute instructions (R code) after each iteration

In English: "**for** each of these values, in this order, execute this set of instructions"

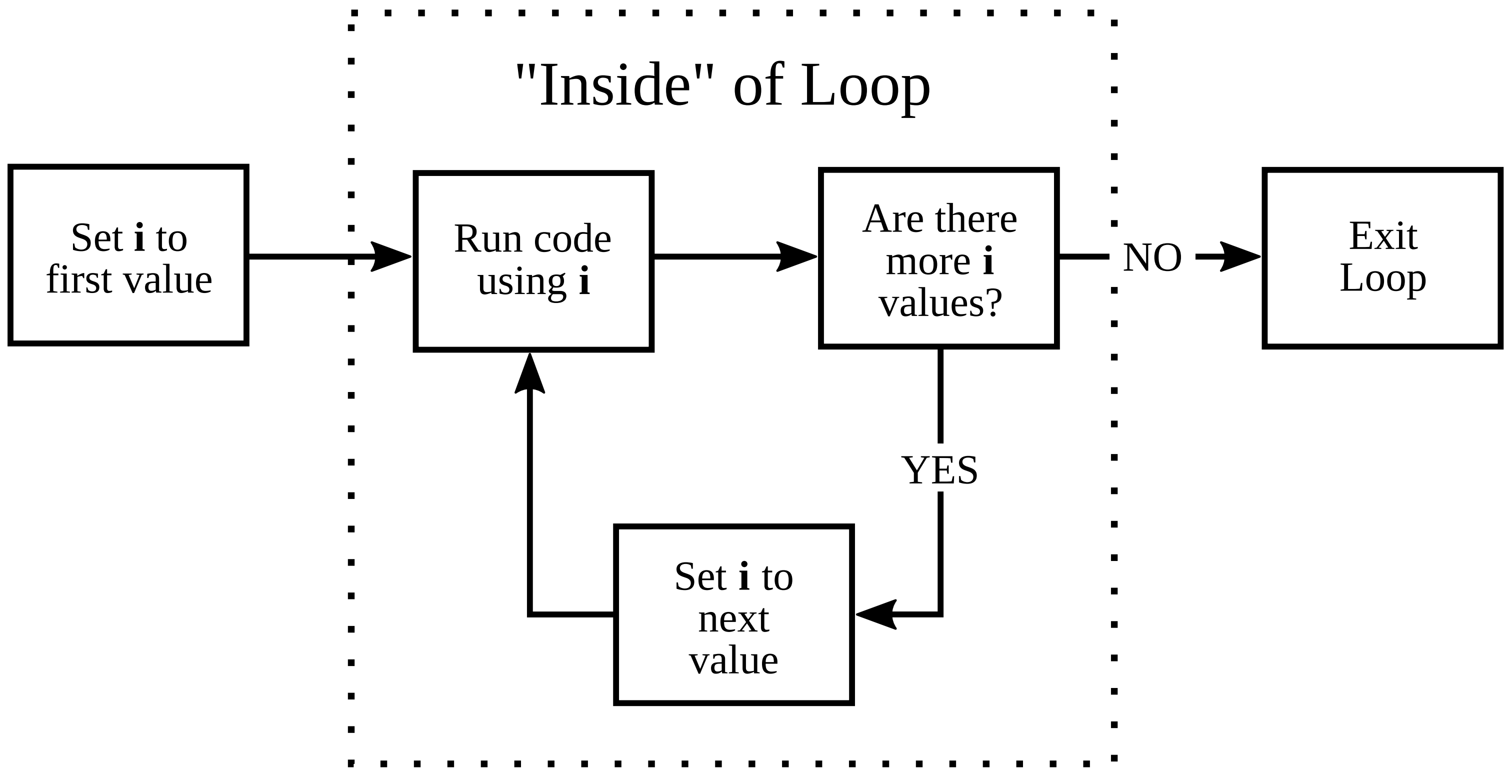
General structure of a `for` loop:

```
for(var in seq){
  expr
}
```

- `var` is an index variable that holds the current value in `seq` (You can call this *whatever* you want! In most cases it is custom to call it `i` but there are meaningful exceptions to this)
- `seq` is a vector of values that you want to iterate over
- `expr` is the R code you want to run for each iteration

`for` Loop: Diagram

Given a set of values:



`for` Loop Example

```r
for(i in 1:10){
  print(i^2)
}
```

```
## [1] 1
## [1] 4
## [1] 9
## [1] 16
## [1] 25
## [1] 36
## [1] 49
## [1] 64
## [1] 81
## [1] 100
```

same as 👉

```r
i <- 1
print(i^2)
```

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

```r
i <- 2
print(i^2)
```

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

```r
i <- 3
print(i^2)
```

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

and so on...

Nested `for` Loops

You can have `for` loops inside of `for` loops inside of `for` loops inside of...

```r
for(i in 1:5){
  for(j in 1:5){
    print(i + j)
  }
}
```

- How many times will `print()` be called?
- What is the first, second, and third output going to be?

```
## 2            (i=1 j=1)
## 3            (i=1 j=2)
## 4            (i=1 j=3)
## 5            (i=1 j=4)
## 6            (i=1 j=5)
## 3            (i=2 j=1)
## 4            (i=2 j=2)
## 5            (i=2 j=3)
## 6            (i=2 j=4)
## 7            (i=2 j=5)
## 4            (i=3 j=1)
## 5            (i=3 j=2)
## 6            (i=3 j=3)
## 7            (i=3 j=4)
## 8            (i=3 j=5)
```

and so on...

There's no limit to how nested you can get[1]:

```r
for(i in 1:10){
  for(j in 50:70){
    for(k in letters){
      for(l in LETTERS){
        for(m in -5:5){
          print(paste(i, j, k, l, m))
        }
      }
    }
  }
}
```

```
## [1] "1 50 a A -5"
## [1] "1 50 a A -4"
## [1] "1 50 a A -3"
## [1] "1 50 a A -2"
```

and so on...

[1] There *are* limits to the computational power you have and to how readable your code is, however. Before using a nested `for` loop, ask yourself if there is a more simple and efficient way of doing what you want

`for` Loop Conventions

- We call what happens in the loop for one particular value one **iteration**
- While you can iterate over any vector, iterating over indices `1:n` is *very* common. `n` might be the length of a vector, the number of rows or columns in a dataframes, or the number of elements in a list
- Common notation: `i` is the object that holds the current value inside the loop
  - If loops are **nested** (one loop inside the other), you will often see `j` and `k` used for the inner loops
  - This notation is similar to indexing in mathematical symbols (e.g., $\sum_{i=1}^{n}$)
- `i` (and `j`, `k`, etc.) are just normal objects. You can use any name you want (e.g., `row` when iterating down rows of a dataframe)

Iterating Over Characters

You don't have to iterate over a numeric vector (although this is most common). You can also iterate over a character vector!

```r
faculty <- c("Corey", "Angela", "Bill", "Mary", "Jane", "Lori")
for(name in faculty){
  print(name)
}
```

```
## [1] "Corey"
## [1] "Angela"
## [1] "Bill"
## [1] "Mary"
## [1] "Jane"
## [1] "Lori"
```

Warning: the *last* value of `var` (in this case `name`) continues to exist outside of the loop. You should *never* name `var` the name of another object.

```r
print(name)
```

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

Pre-allocation

Usually in a `for` loop you are not just printing output, but want to store results from calculations in each iteration somewhere

To do that, figure out what you want to store and **pre-allocate** an object of the right size as a placeholder (typically filled with `NA`)

```r
results <- rep(NA_real_, 10000) # Vectors with 10,000 NAs (numeric)

for(i in 1:10000){
  results[i] <- i + i^2 + i^3
}
head(results)
```

```
## [1]   3  14  39  84 155 258
```

```r
# Check if there are any NAs still in results
any(is.na(results))
```

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

You don't need to pre-allocate a vector. Instead you can instantiate (i.e., create) an empty vector and fill it in as you go:

```r
results <- c()

for(i in 1:10000){
  results[i] <- i + i^2 + i^3
}
head(results)
```

```
## [1]   3  14  39  84 155 258
```

You can also use the `append()` function to add values to the end (or the `after`th value) of the vector[1]

```r
for(i in 1:10000){
  results <- append(results, i + i^2 + i^3)
}
```

[1] See `?append` to learn more about the function

Warning: Although it likely won't make much of a difference for you, pre-allocating a vector is substantially faster than filling in an empty vector

Let's see how long it takes for R to fill in 100 million values in an empty and a pre-allocated vector

```r
n <- 100000000

system.time({
  vec <- rep(NA_real_, n)

  for(i in 1:n){
    vec[i] <- i
  }
})
```

```
##    user  system elapsed 
##   5.405   0.248   5.654
```

```r
system.time({
  vec <- c()

  for(i in 1:n){
    vec[i] <- i
  }
})
```

```
##    user  system elapsed 
##  26.658   2.496  29.156
```

`setNames()`

Using the `setNames()` function, you can pre-allocate a *named* vector:

```
vec <- setNames(object = rep(NA_real_, 10), # vector
                nm = paste0("elem", 1:10))  # names

print(vec)
```

```
##  elem1  elem2  elem3  elem4  elem5  elem6  elem7  elem8  elem9 el
##     NA     NA     NA     NA     NA     NA     NA     NA     NA
```

Debugging `for` Loops

Let's say we want to take the mean across all columns in the `swiss` dataset:

```r
swissmeans <- c()

for(i in 1:ncol(swiss)){ # vector 1, 2, 3,... ncols in swiss
  swissmeans[i] <- mean(swiss[, i], na.rm = T)
}
```

```
## Warning in mean.default(swiss[, i], na.rm = T): argument is not n
## logical: returning NA
```

☝️ This warning tells us that our call to `mean()` didn't work for one iteration, but we don't know *which* iteration. Understanding warnings and errors inside loops becomes even more challenging when you have a lot of instructions inside the loop!

One way to debug a `for` loop is to simulate every iteration of the loop yourself. In this case, this would look like:

```r
i <- 1 # set i manually
mean(swiss[, i], na.rm = T) # run suspect code
```

```
## [1] 70.14255
```

```r
i <- 2 # increment i manually
mean(swiss[, i], na.rm = T) # run suspect code
```

```
## [1] 50.65957
```

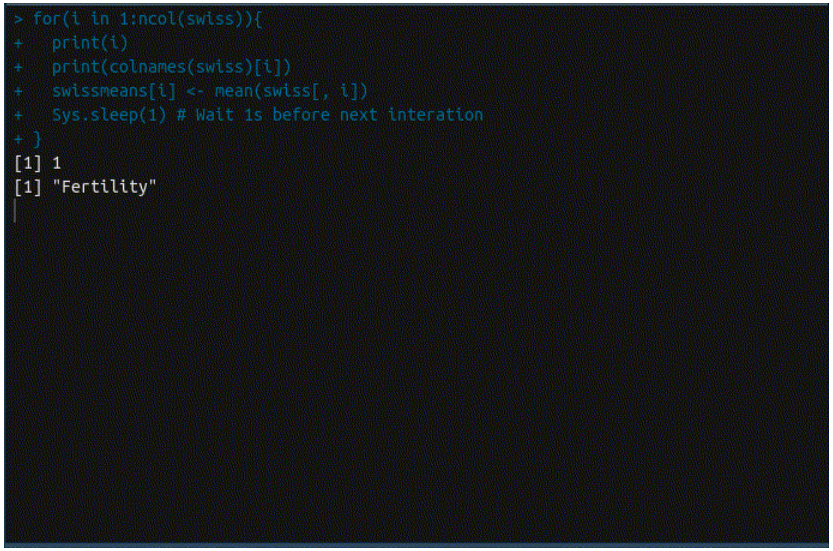
This is very time consuming, especially when you have many iterations and/or lots of R code to execute

Debug with `print()` and `Sys.sleep()`

Instead of running through the loop manually, use `print()` to print the current iteration and any variables you think might be responsible for the bug

Use `Sys.sleep()` if you want to slow the loop down during debugging to give you more time to process what's happening:

```r
for(i in 1:ncol(swiss)){
  print(i)
  print(colnames(swiss)[i])
  swissmeans[i] <- mean(swiss[, i])
  Sys.sleep(1) # Wait 1s before next interation
}
```



```
> for(i in 1:ncol(swiss)){
+   print(i)
+   print(colnames(swiss)[i])
+   swissmeans[i] <- mean(swiss[, i])
+   Sys.sleep(1) # Wait 1s before next interation
+ }
[1] 1
[1] "Fertility"
```

UNIVERSITY OF WASHINGTON

Example: LOOCV

Leave-one-out cross-validation (LOOCV) is a method used to evaluate how well a model will predict a *new* observation (i.e., not an observation in the "training" data, which is characterized by the residual term)

To perform LOOCV:

1. Remove one observation from the data
2. Fit your model
3. See how well the model predicts the removed observation
4. Repeat with a newly removed observation for *n observations*

Simulate data for example:

```r
set.seed(98195)
n <- 300

# tibbles are like dataframes (we will cover them later this quarter)
dat_sim <- tibble(x = rnorm(n, mean = 5, sd = 4),
                  z = x + rnorm(n, mean = 0, sd = 10),
                  y = 2 + (-0.5*x) + (.5*x^2) + (-0.5*z) + rnorm(n))
```

Visualizing `dat_sim`

```r
# Empty column to hold the results
dat_sim$ypred <- rep(NA_real_, nrow(dat_sim))

# Conduct LOOCV
for(i in 1:nrow(dat_sim)){
  # Estimate linear model
  fit <- lm(y ~ x + I(x^2) + z,
            data = dat_sim[-i, ]) # Remove ith row of dat_sim

  # Predict y of removed observation
  # (equivalent to just plugging in the numbers ourselves)
  dat_sim$ypred[i] <- predict(fit, newdata = dat_sim[i, c("x", "z")])
}

head(dat_sim)
```

```
## # A tibble: 6 x 4
##       x     z     y ypred
##   <dbl> <dbl> <dbl> <dbl>
## 1  1.41 15.8  -6.03 -5.39
## 2 10.8   5.41 51.7  51.7
## 3  7.12 12.8  17.2  17.4
## 4  9.99  5.36 44.6  44.1
## 5 10.5  17.8  42.1  42.8
## 6  4.48  6.83  5.67  6.41
```

By definition, the residuals from `lm()` will be smaller than from `LOOCV` on the same dataset. Let's check to make sure this is true:

```
#  lm()
pred_lm <- predict(lm(y ~ x + I(x^2) + z,
                      data = dat_sim))

mean((pred_lm - dat_sim$y)^2)
```

```
## [1] 1.062459
```

```
# LOOCV
mean((dat_sim$ypred - dat_sim$y)^2)
```

```
## [1] 1.093004
```

Example: K-Fold Cross Validation

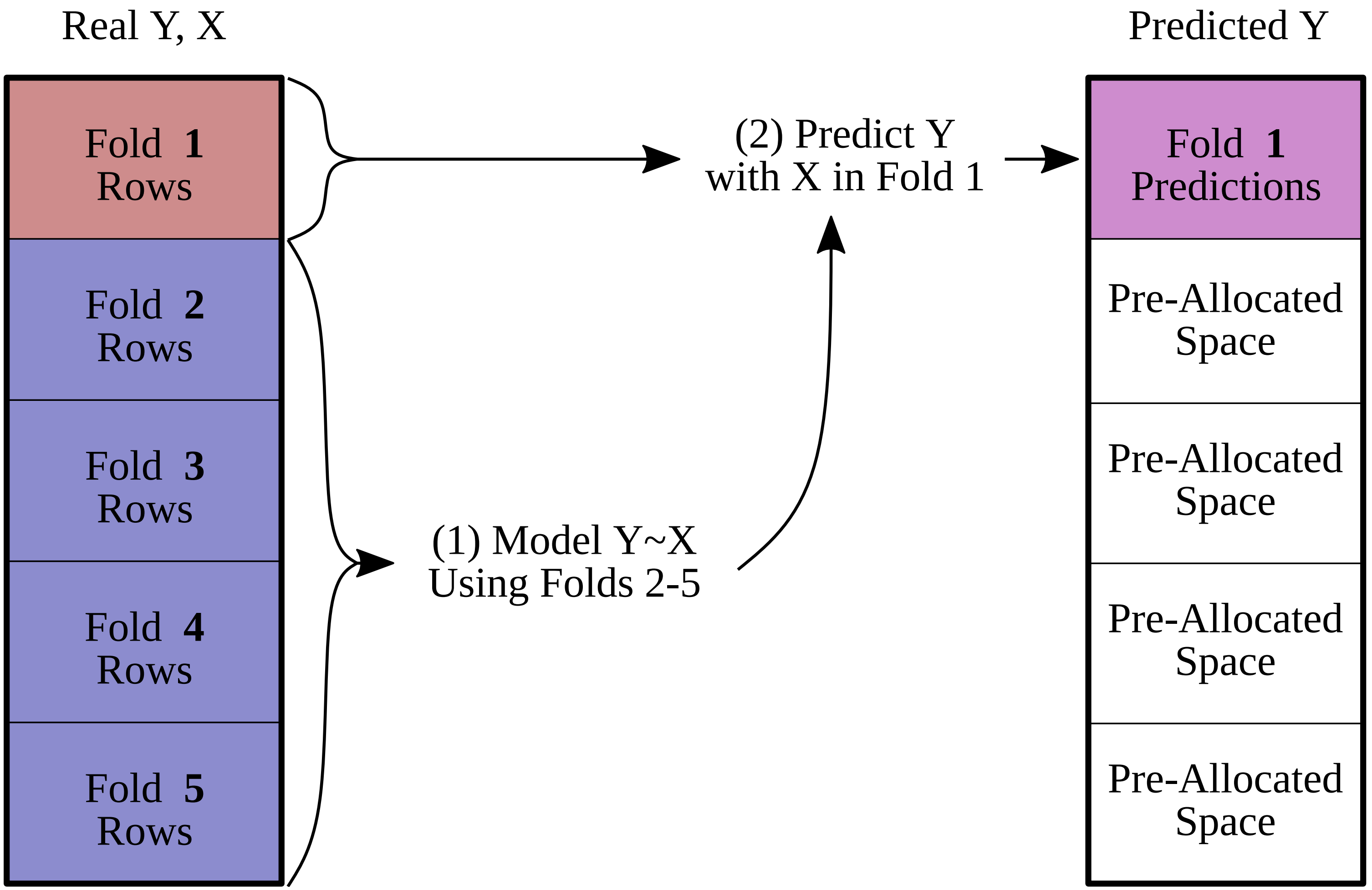
K-fold cross validation involves running your model on random subsets of your data and using the remaining data to estimate how well the model performed

LOOCV is a form of k-fold cross validation where *K* is the number of rows

1. Split your data into *K* **folds**
2. For each fold $i = 1, \dots, K$:
   - Fit the model to all the data *except* that in fold *i*
   - Make predictions for the omitted data in fold *i*
3. Calculate accuracy (mean squared erro or however you'd like)

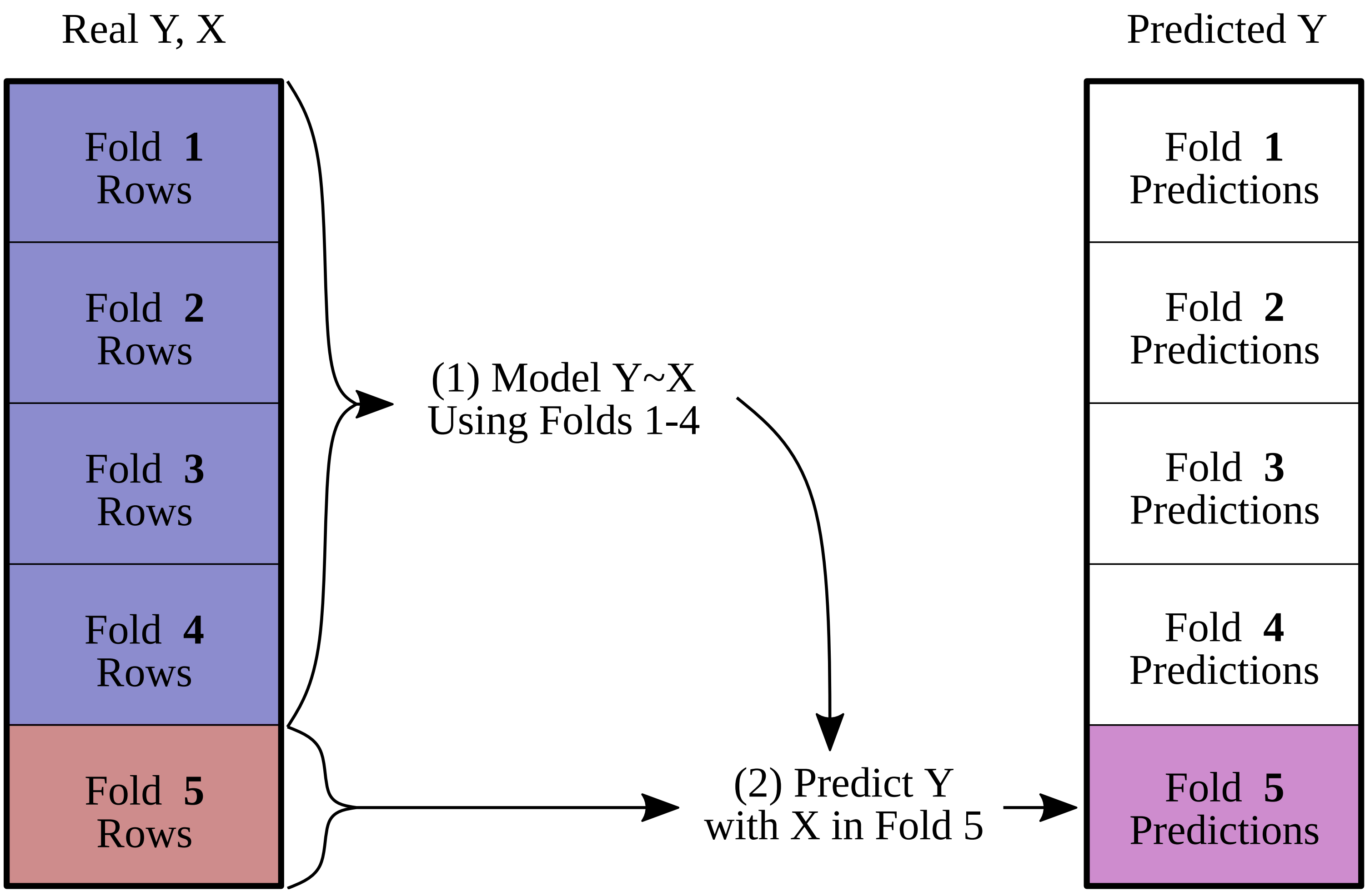
A model that fits well has a *lower* mean squared error

Iteration **1**



Real Y, X
Predicted Y
Fold **1** Rows
Fold **2** Rows
Fold **3** Rows
Fold **4** Rows
Fold **5** Rows
(2) Predict Y with X in Fold 1
(1) Model Y~X Using Folds 2-5
Fold **1** Predictions
Pre-Allocated Space
Pre-Allocated Space
Pre-Allocated Space
Pre-Allocated Space

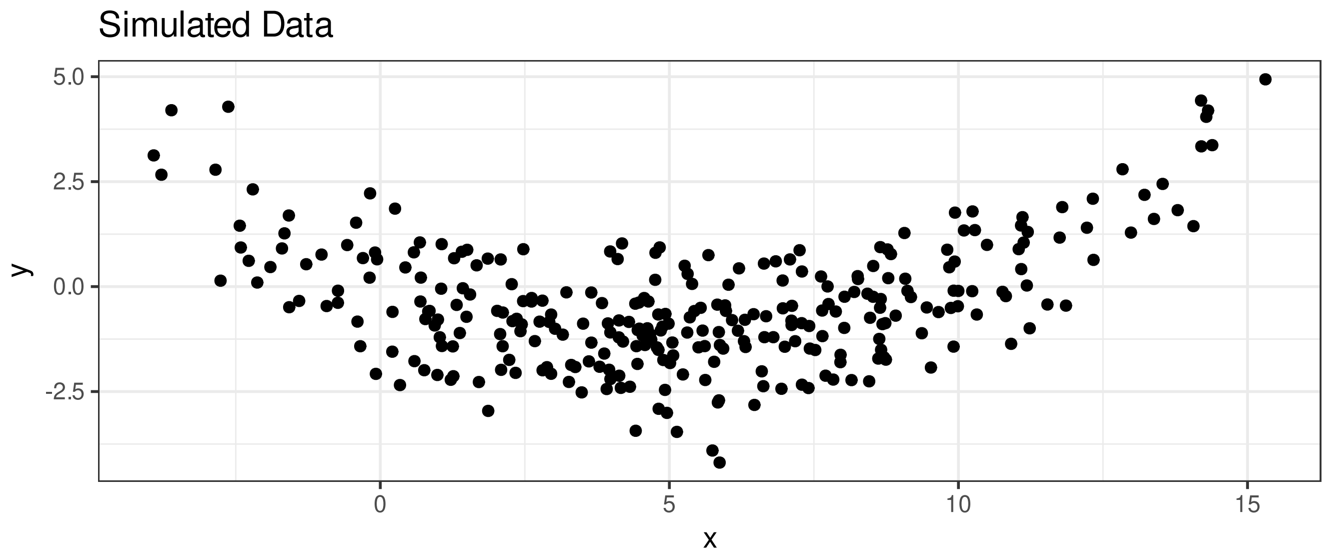
Iteration 5


Real Y, X
Predicted Y
Fold 1 Rows
Fold 2 Rows
Fold 3 Rows
Fold 4 Rows
Fold 5 Rows
(1) Model Y~X Using Folds 1-4
(2) Predict Y with X in Fold 5
Fold 1 Predictions
Fold 2 Predictions
Fold 3 Predictions
Fold 4 Predictions
Fold 5 Predictions

Let's simulate some fake data for this using the `rnorm()` function to generate random values from a normal distribution.

```
set.seed(98195)
n <- 300
dat_sim <- tibble(x = rnorm(n, mean = 5, sd = 4),
                  y = -0.5 * x + 0.05 * x^2 + rnorm(n, sd = 1))
```

This generates a dataframe of 300 observations where `y` is dependent on `x`, with some uncorrelated, normally-distributed residual (from `rnorm()`).



Candidate Regression Models

Let's say we want to consider several different regression models to draw trendlines through these data:

- **Intercept Only:** draw a horizontal line that best fits the `y` values.

$$\hat{y}_i = \beta_0$$

- **Linear Model:** draw a line that best fits the `y` values as a function of `x`.

$$\hat{y}_i = \beta_0 + \beta_1 x_i$$

- **Quadratic Model:** draw a quadratic curve that best summarizes the `y` values as a function of `x`.

$$\hat{y}_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2$$

- **Cubic Model:** draw a cubic curve that best summarizes the `y` values as a function of `x`.

$$\hat{y}_i = \beta_0 + \beta_1 x_i + \beta_2 x_i^2 + \beta_3 x_i^3$$

Pre-Allocating for CV

Let's make a *named character vector* for the formulas we'll use in `lm()`:

```
models <- c("intercept only" = "y ~ 1",
            "linear" = "y ~ x",
            "quadratic" = "y ~ x + I(x^2)",
            "cubic" = "y ~ x + I(x^2) + I(x^3)")
```

Let's also split the data into $K = 10$ folds. We will make a new dataframe to hold the data and sampled fold numbers that we'll add predictions to later.

```
K <- 10
CV_pred <- dat_sim
CV_pred$fold <- sample(rep(1:K, length.out = nrow(CV_pred)),
                        replace = FALSE)
CV_pred[ , names(models)] <- NA_real_
head(CV_pred)
```

```
##           x          y fold intercept only linear quadratic cubic
## 1  1.410985  0.8298248    6             NA     NA        NA    NA
## 2 10.762657 -0.1251665    9             NA     NA        NA    NA
## 3  7.120262 -0.4582323    8             NA     NA        NA    NA
## 4  9.989839 -0.4680869    7             NA     NA        NA    NA
## 5 10.493456  0.9938052    8             NA     NA        NA    NA
## 6  4.480082 -1.0018591    7             NA     NA        NA    NA
```

Double-Looping for CV

Next, let's loop *over* each model (`mod`), and *within* each model loop over each fold (`k`) to fit the model and make predictions.

```r
for(mod in names(models)) {
    for(k in 1:K) {

        # Fit model to data not in fold
        fit <- lm(formula(models[mod]),
                  data = CV_pred[CV_pred$fold != k, ])

        # Predict on data in fold
        CV_pred[CV_pred$fold == k, mod] <- predict(fit, newdata = CV_pred[CV_pred$
    }
}
```

Note the models are fit *without* the fold rows, but prediction is done on *only the left-out fold rows*.

Which Model Fit Best?

Let's write another loop to compute the mean squared error of these CV predictions

The squared error is equal to the difference between the observed values and predicted values squared. The MSE is the mean of all the squared errors of each prediction.

```r
CV_MSE <- setNames(numeric(length(models)), names(models))
for(mod in names(models)) {
    pred_sq_error <- (CV_pred$y - CV_pred[[mod]])^2
    CV_MSE[mod] <- mean(pred_sq_error)
}
print(CV_MSE)
```

```
## intercept only         linear      quadratic          cubic
##       2.196284       2.164999       1.060667       1.070219
```

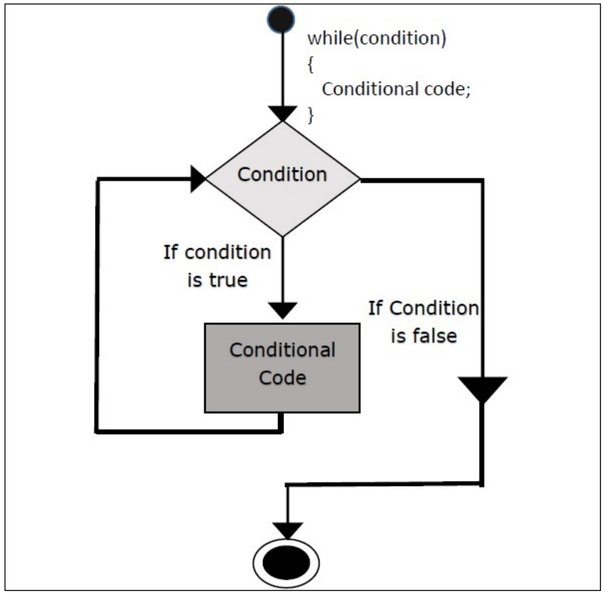
Based on these results, which model would you choose?

`while` Loop

`while` loops repeat a set of instructions (R code) **while** a certain condition is met

```
while(cond){
  expr
}
```

- `cond` is a logical statement that evaluates to `TRUE`/`FALSE`
- `expr` is the R code you want to run iteratively



`while` Loop

```r
# Initialize x
x <- 0

while(x < 10){
  # Increment x
  x <- x + 2

  print(x)
}
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
```

Notice that, unlike the `for` loop, we don't necessarily know how long the `while` loop will run. This is exactly when a `while` loop is preferred!

`while` Loop: Example

Let's say we want to know how much times a number can be cut in half until it is less than or equal to 1:

```r
count <- 0
number <- 32
while(number > 1){
  print(number)
  number <- number / 2
  count <- count + 1
}
```

```
## [1] 32
## [1] 16
## [1] 8
## [1] 4
## [1] 2
```

```r
print(paste(count, "times!"))
```

```
## [1] "5 times!"
```

```r
count <- 0
number <- 908345903485304
while(number > 1){
  print(number)
  number <- number / 2
  count <- count + 1
}
```

```
## [1] 9.083459e+14
## [1] 4.54173e+14
## [1] 2.270865e+14
## [1] 1.135432e+14
## ...
```

```r
print(paste(count, "times!"))
```

```
## [1] "50 times!"
```

`while` Loops: A Warning

Be careful when writing a `while` loop to make sure the condition will be met eventually!

This will run forever:

```r
while(TRUE){
  print(😭)
}
```

```
😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭
😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭
😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭
😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭
😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭
😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭
😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭
😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭
😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭😭…
```

`repeat` Loop

`repeat` loops repeat a set of instructions until a certain condition is met

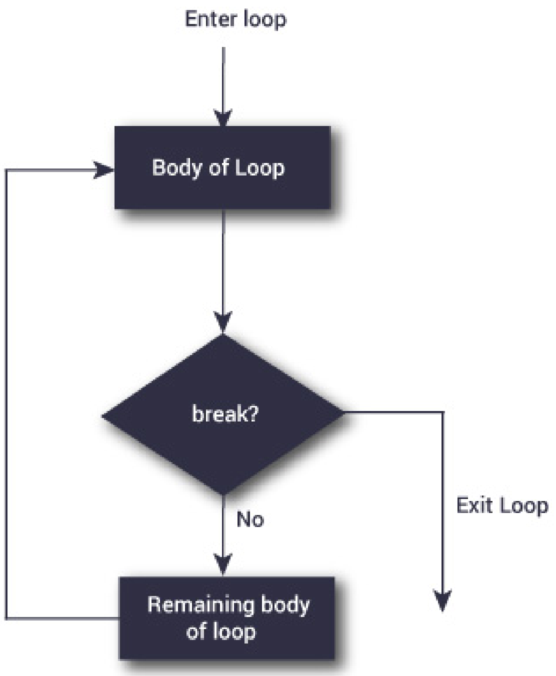
`repeat` loops differ from `while` loops because the *condition* is evaluated at the end (rather than the begining) of the loop

---

```
repeat {
  expr
  if(condition){
    break
  }
}
```

Warning: even more so than the `while` loop, a `repeat` loop will run forever until you tell it to stop!

To stop the loop, use the `break` statement



`repeat` Loop: Example

Using a `repeat` loop, find the factorial of a given number (the product of all integers from 1 to *number*)

```r
# Initialize objects
number <- 30
i <- 2 # counter
res <- 1 # holds result

repeat{

  res <- res*i

  # If 1*2*3...n, stop
  if(i == number){
    break
  }

  # Increment counter
  i <- i + 1
}

print(res)
```

```
## [1] 2.652529e+32
```

`next`

`next` allows you to immediately skip to the **next** iteration of a loop without executing the code below. You can use this in all three types of loops!

```r
for(i in 1:15){
  # Notice that the entire if()
  # statement is on one line.
  # This is okay when it is
  # simple like this
  if(i %in% 6:10) next
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
```

```r
i <- 0
while(i < 15){
  i <- i + 1
  if(i %in% 6:10) next
  print(i)
}
```

```r
i <- 0
repeat{
  i <- i + 1
  if(i %in% 6:10) next
  print(i)
  if(i == 15) break
}
```

Vectorization

What is Vectorization?

One of the many things that makes R a unique and useful language for working with data is the concept of **vectorization**

Consider a simple data cleaning problem where you need to convert all 0s to 1s and all 1s to 0s (e.g., to change the reference group in your linear model). To do this, all you need to do is subtract each observation from 1. But what if you have hundreds, thousands, or even millions of observations?

Good thing we just learned about loops!

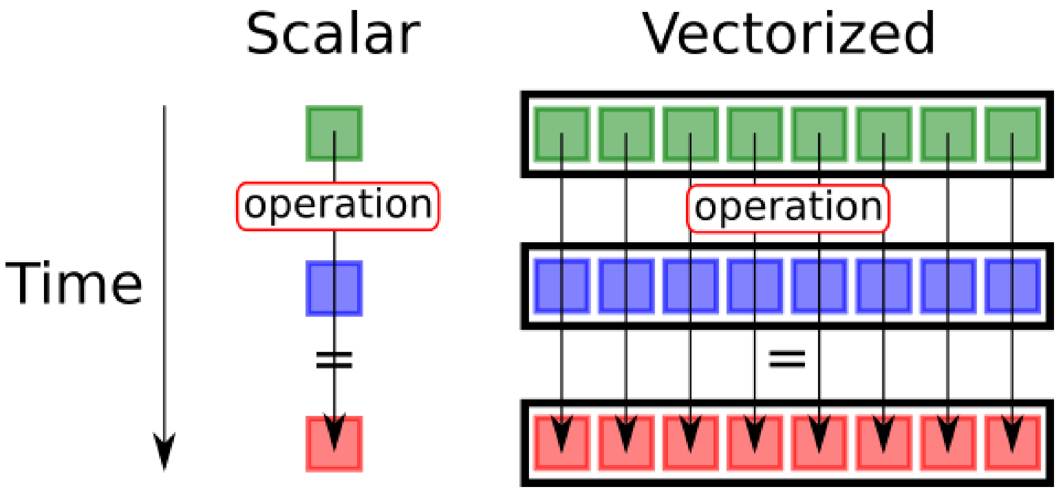
```r
# Create vector of 100,000,000 0s and 1s
vec <- sample(0:1, 100000000, TRUE)

system.time(
  for(i in 1:length(vec)){
    vec[i] <- 1 - vec[i]
  }
)
```

```
##    user  system elapsed 
##   6.897   0.376   7.273
```

👈 This takes forever!

Vectorization Wins

Instead of using a loop to do this task, we can use R's vector addition, which is a **vectorized** function



```r
system.time(
  1 - vec
)
```

```
##    user  system elapsed 
##   0.310   0.208   0.518 
```

```r
system.time(
  for(i in 1:length(vec)){
    vec[i] <- 1 - vec[i]
  }
)
```

```
##    user  system elapsed 
##   8.308   0.280   8.592 
```

Vectorization Examples

`rowSums()`, `colSums()`, `rowMeans()`, `colMeans()` return a vector of sums or means over rows or columns of data (these are very useful for constructing scale scores!)

```
my_matrix <- matrix(1:12, nrow = 3, ncol = 4, byrow = TRUE)
print(my_matrix)
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    5    6    7    8
## [3,]    9   10   11   12
```

```
rowSums(my_matrix)
```

```
## [1] 10 26 42
```

More Vectorization Examples

`cumsum()`, `cumprod()`, `cummin()`, `cummax()` return a vector of cumulative quantities

```
cumsum(1:10)
```

```
##  [1]  1  3  6 10 15 21 28 36 45 55
```

```
cummin(c(3:1, 2:0, 4:2))
```

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

`pmax()` and `pmin()` take a matrix or set of vectors and return the min or max for each position (after recycling)

```
pmax(c(0, 3, 4),
     c(1, 1, 1),
     c(2, 2, 2))
```

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