Week 3: Control Flow in R

POP77001 Computer Programming for Social Scientists

Tom Paskhalis

Overview

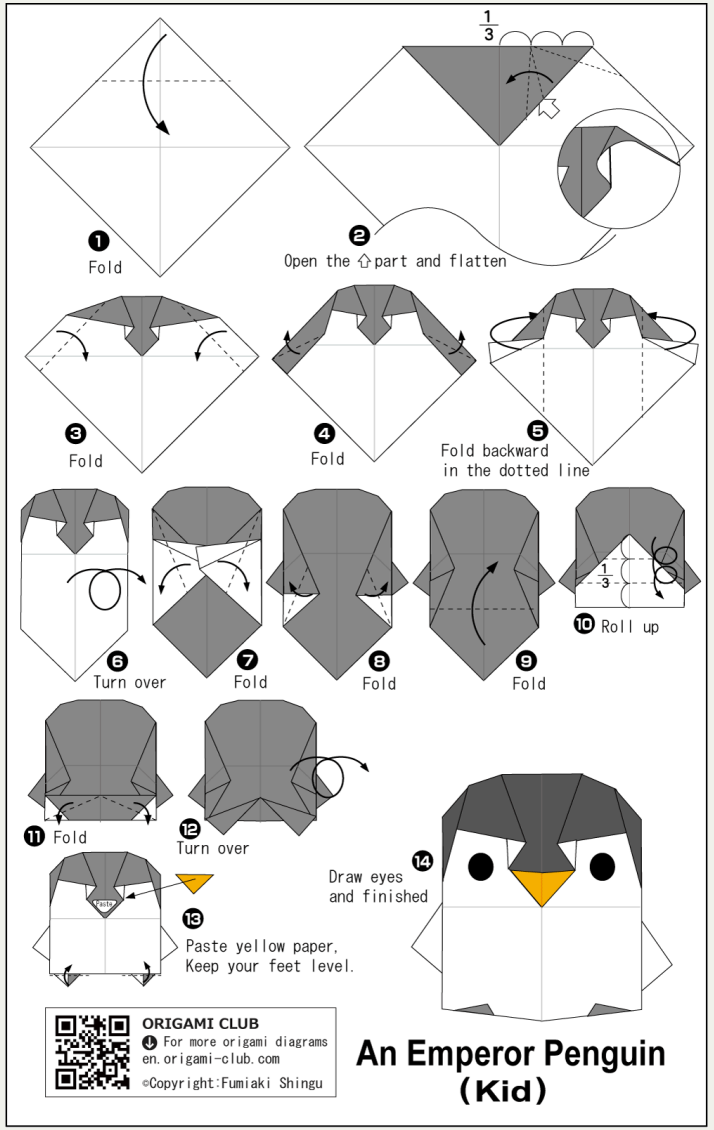
- Straight-line and branching programs
- Algorithms
- Conditional statements
- Loops and Iteration

Algorithm

Algorithm

- *Finite list of well-defined instructions that take input and produce output.*
- Consists of a sequence of simple steps that start from input, follow some control flow and have a stopping rule.

Algorithm Example

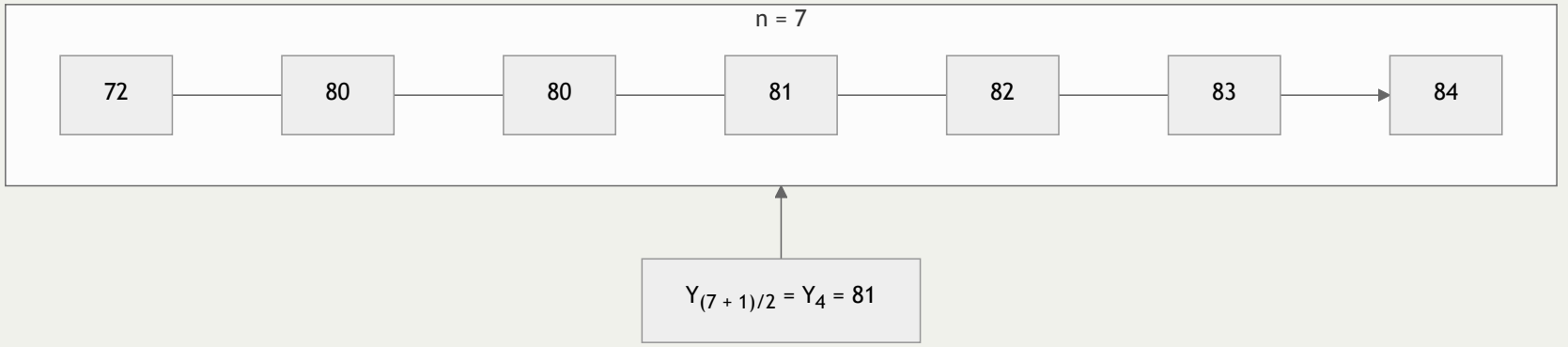


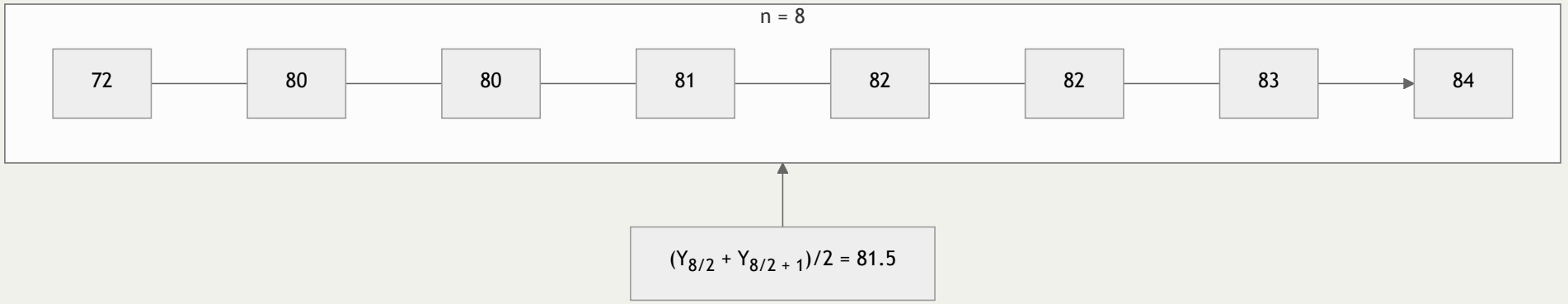
Origami Club

Median

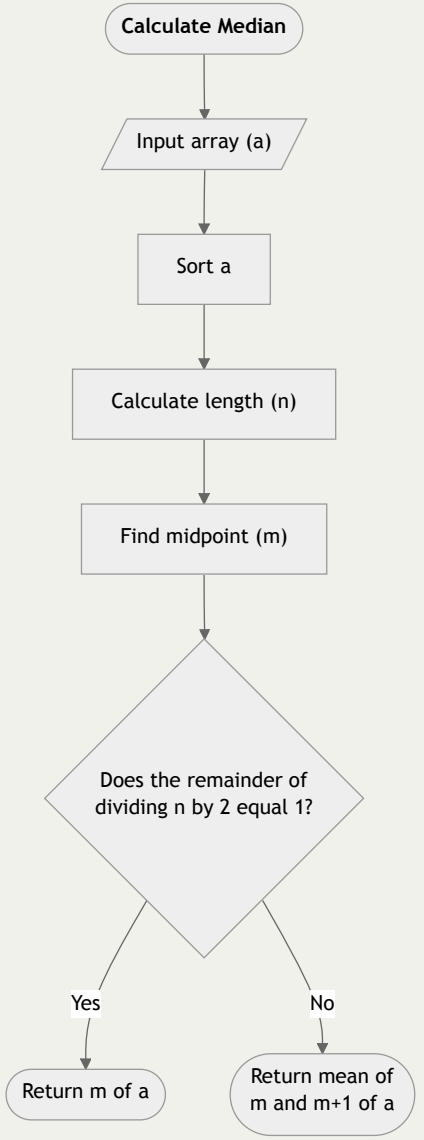
- The value that falls in the middle of an ordered sample.

$$Median = \begin{cases} Y_{(n+1)/2} & \text{when n is odd} \\ \frac{1}{2}(Y_{(n/2} + Y_{n/2+1}) & \text{when n is even} \end{cases}$$

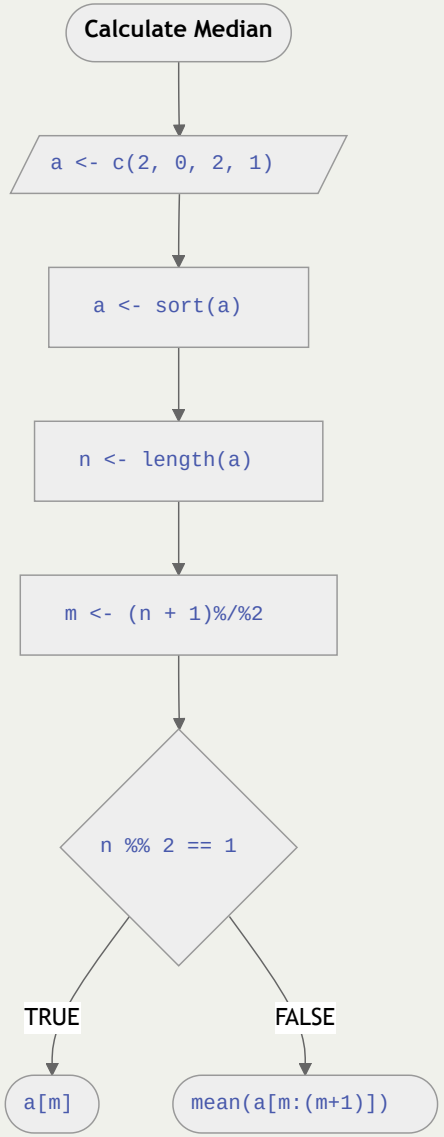




Algorithm Flowchart



Algorithm Flowchart (R)



Calculate Median

```r
a <- c(2, 0, 2, 1) # Input vector (1-dimensional array)
a <- sort(a) # Sort vector
a
```

```
[1] 0 1 2 2
```

```r
# Calculate length of vector 'a'
n <- length(a)
n
```

```
[1] 4
```

```r
# Calculate mid-point, %/% is operator for integer division
m <- (n + 1) %/% 2
m
```

```
[1] 2
```

```r
# Check whether the number of elements is odd,
# %% (modulo) gives the remainder of division
n %% 2 == 1
```

```
[1] FALSE
```

```r
mean(a[m:(m+1)])
```

```
[1] 1.5
```

Control Flow

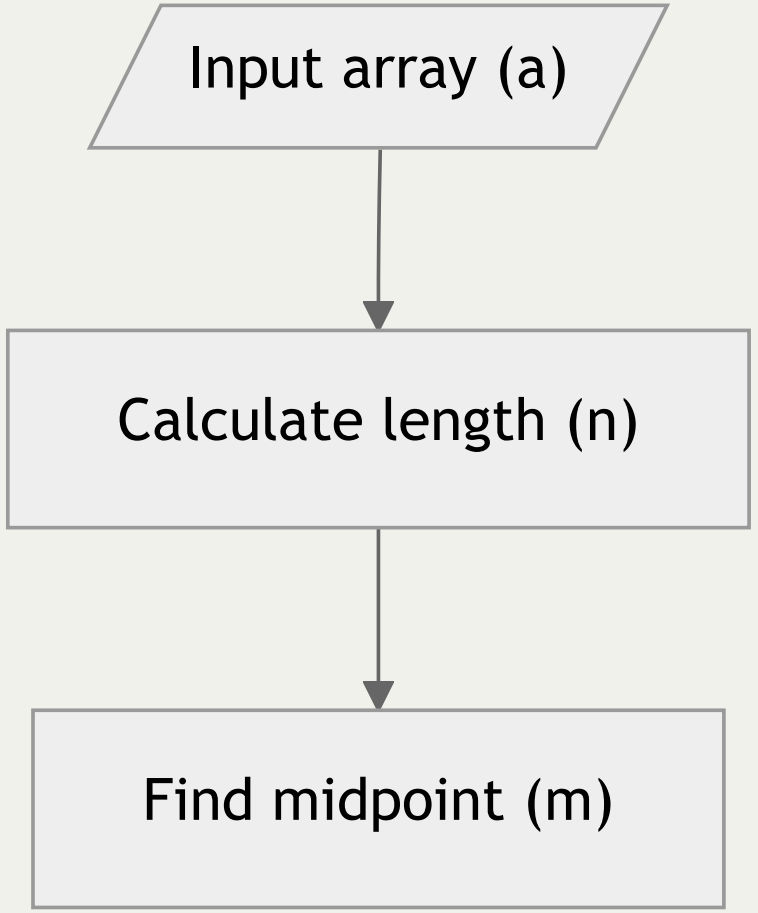
Control Flow in R

- **Control flow** is the order in which statements are executed or evaluated
- Main ways of control flow in R:
  - **Branching** (conditional) statements (e.g. `if`)
  - **Iteration** (**loops**) (e.g. `for`)
  - **Function calls** (e.g. `length()`)

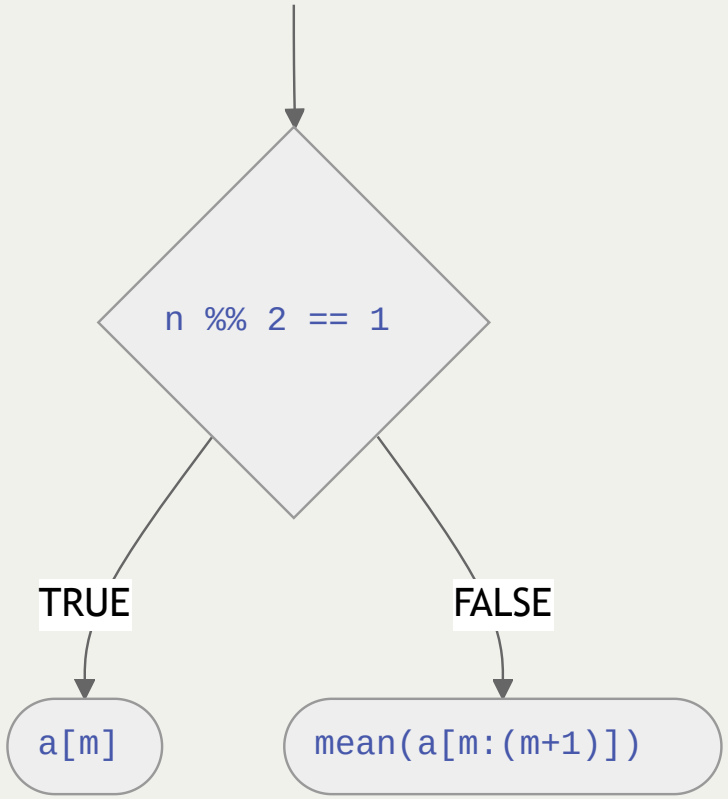
Extra

R documentation on control flow

Straight-Line Programs

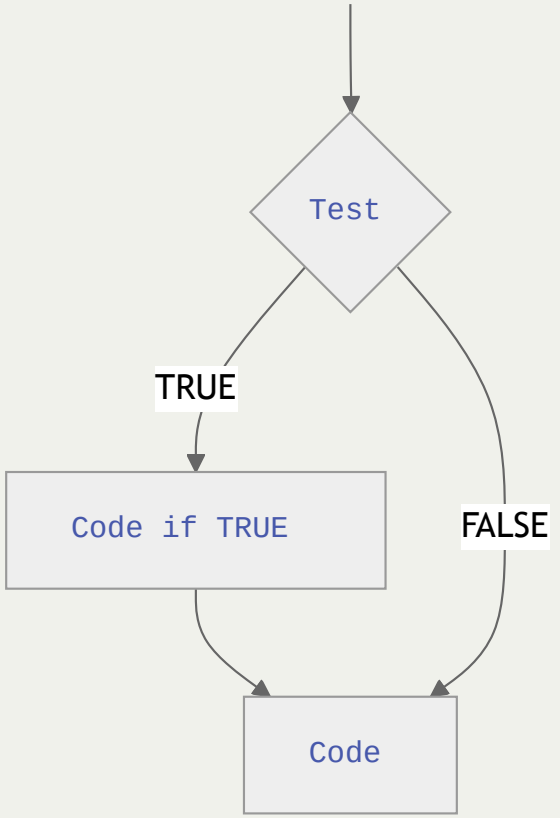


Branching Programs



Conditional Statements

Simple Conditional Statement



Basic Conditional Statement: `if`

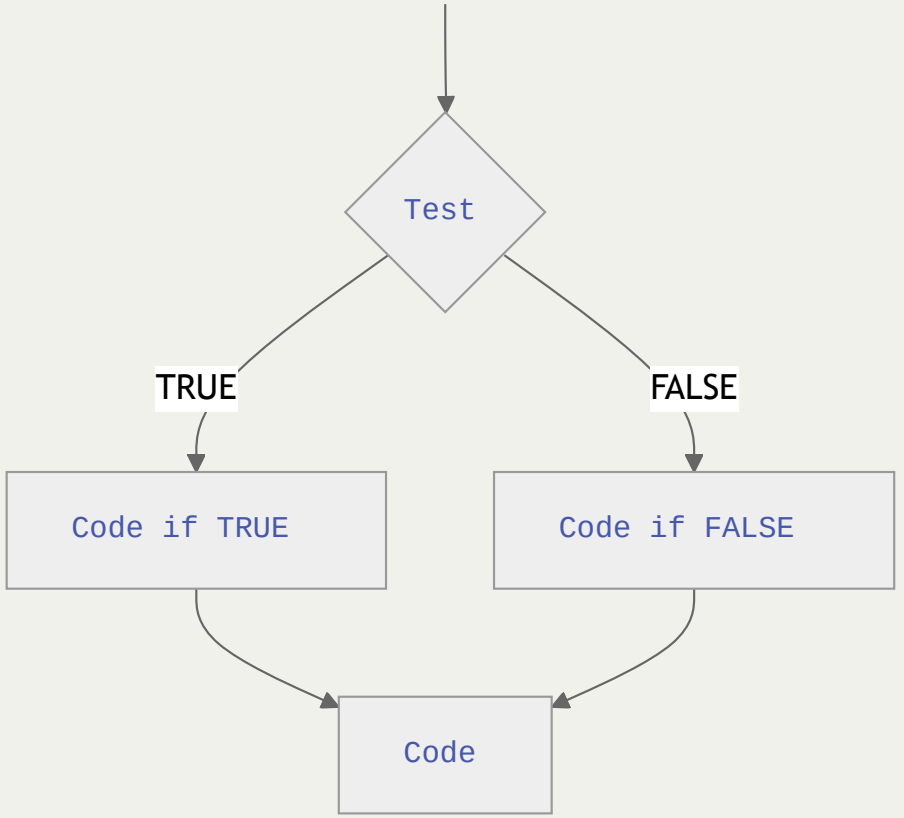
- `if` - defines condition under which some code is executed

```
if (<boolean_expression>) {
  <some_code>
}
```

```r
a <- c(2, 0, 2, 1, 100)
a <- sort(a)
n <- length(a)
m <- (n + 1) %/% 2
if (n %% 2 == 1) {
  a[m]
}
```

```
[1] 2
```

Complex Conditional Statements



if - else

- `if - else` - defines both condition under which some code is executed and alternative code to execute

```
if (<boolean_expression>) {
  <some_code>
} else {
  <some_other_code>
}
```

```
a <- c(2, 0, 2, 1)
a <- sort(a)
n <- length(a)
m <- (n + 1) %/% 2
if (n %% 2 == 1) {
  a[m]
} else {
  mean(a[m:(m+1)])
}
```

```
[1] 1.5
```

if - else if - else

- `if - else if - ... - else` - defines both condition under which some code is executed and several alternatives

```
if (<boolean_expression>) {
  <some_code>
} else if (<boolean_expression>) {
  <some_other_code>
} else if (<boolean_expression>) {
...
...
} else {
  <some_more_code>
}
```

```
mark <- 71
if (mark >= 70) {
  grade <- "I"
} else if (mark >= 60) {
  grade <- "II.1"
} else if (mark >= 50) {
  grade <- "II.2"
} else {
  grade <- "F"
}
grade
```

```
[1] "I"
```

Optimising Conditional Statements

- Parts of conditional statement are evaluated sequentially, so it makes sense to put the most likely condition as the first one:

```
# Check the OS (Operating System) of the computer
platform <- .Platform$OS.type
if (platform == "windows") {
  file_sep <- "\\" # Technically R uses '/' on Windows machines
} else if (platform == "unix") {
  file_sep <- "/"
} else {
  file_sep <- NA
}
file_sep
```

```
[1] "/"
```

Nesting Conditional Statements

- Conditional statements can be nested within each other
- But consider code legibility 📜, modularity ⚙ and speed 🏎

```r
mark <- 65
# Optimising for the most likely condition
if (mark >= 50 && mark < 70) {
  if (mark >= 60) {
    grade <- "II.1"
  } else {
    grade <- "II.2"
  }
} else {
  if (mark >= 70) {
    grade <- "I"
  } else {
    grade <- "F"
  }
}
```

`ifelse()` function

- R also provides a vectorized version of `if - else` construct.
- It takes a vector as an input and returns another vector as an output.
- The 1st argument is an expression that evaluates to a boolean vector.

```
ifelse(<boolean_expression>, <if_true>, <if_false>)
```

```
1  num <- 1:10
2  num
```

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

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

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

Condition

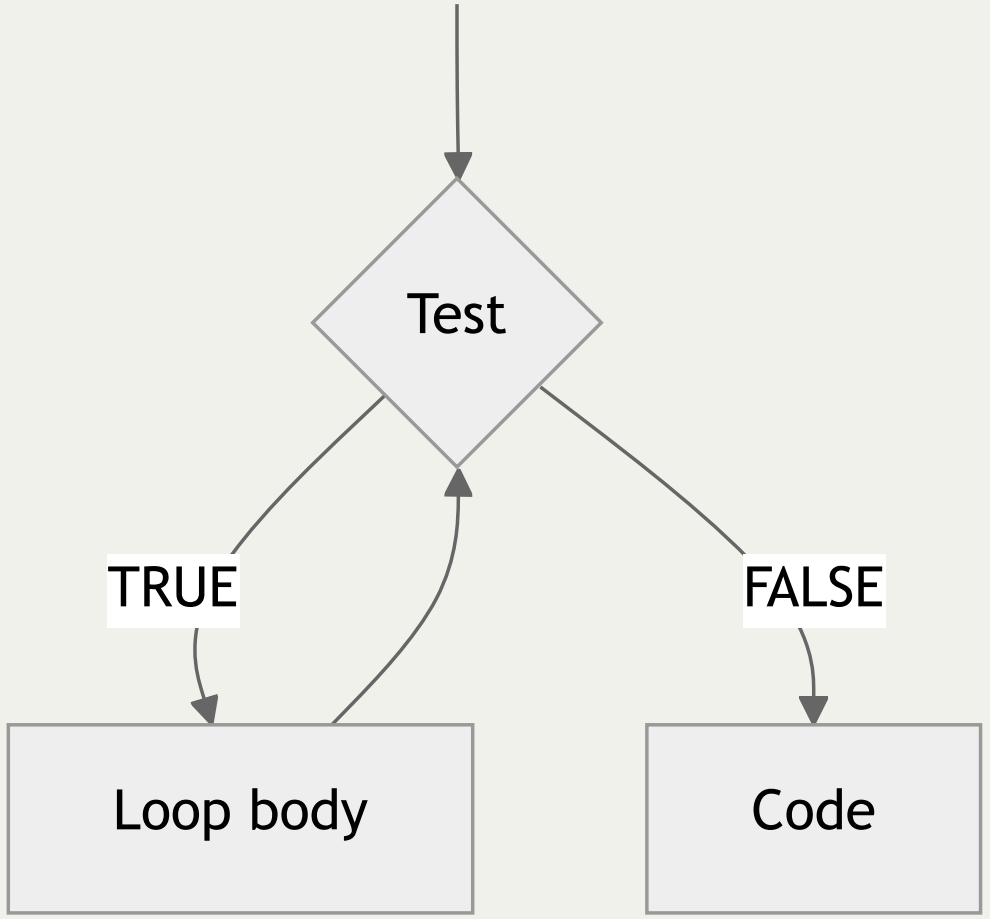
- The condition must evaluate to a single `TRUE` or `FALSE`.
- Was a warning prior to R `4.2.0`.

```
1 if (c(TRUE, FALSE)) 1
```

```
Error in if (c(TRUE, FALSE)) 1: the condition has length > 1
```

Iteration

Loop



while

- `while` defines a condition under which some code (loop body) is executed repeatedly.

```
while (<boolean_expression>) {
  <some_code>
}
```

```r
# Calculate a factorial with decrementing function
# E.g. 5! = 1 * 2 * 3 * 4 * 5 = 120
x <- 5
factorial <- 1
while (x > 0) {
  factorial <- factorial * x
  x <- x - 1
}
factorial
```

```
[1] 120
```

for

- `for` defines elements and sequence over which some code is executed iteratively.

```
for (<element> in <sequence>) {
  <some_code>
}
```

```
1 x <- seq(5)
2 factorial <- 1
3 for (i in x) {
4   factorial <- factorial * i
5 }
6 factorial
```

```
[1] 120
```

Iteration & Conditional Statements

Iterations can be combined with conditional statements to create more complex control flows.

```r
# Find maximum value in a vector with exhaustive enumeration
v <- c(3, 27, 9, 42, 10, 2, 5)
max_val <- v[1]
for (i in v[2:length(v)]) {
  if (i > max_val) {
    max_val <- i
  }
}
max_val
```

```
[1] 42
```

Iteration Sequences

Collections

- Typically, we iterate over existing collections of elements.
- E.g. vectors, lists, data frames, etc.
- But sometimes we need to first create a collection to iterate over.
- Common usecases include generating sequences of indices and preallocating containers.

Generating Sequences with :

- Colon operator `:` allow to generate sequences of integers.
- We saw those in subsetting, but they are also useful in iteration.

`<from>:<to>`

```
1 1:5
```

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

```
1 # Note that R automatically infers the
2 # direction of the sequence (increasing or decreasing)
3 1:-5
```

```
[1]  1  0 -1 -2 -3 -4 -5
```

Generating Sequences with `seq()`

- `seq()` function that we encountered in subsetting can be used in looping.
- It is a generalisation of `:` that allows to specify the step size.
- As well as its cousins: `seq_len()` and `seq_along()`

```
seq(<from>, <to>, <by>)
seq_len(<length>)
seq_along(<object>)
```

```
s <- seq(1, 20, 2)
s
```

```
[1]  1  3  5  7  9 11 13 15 17 19
```

```
# seq_len() is equivalent to seq(1, length(<object>))
seq_len(length(s))
```

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

```
# seq_along() is equivalent to seq(along.with = <object>)
seq_along(s)
```

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

```
# The sequence that you are supplying to seq_along()
# does not have to be numeric
seq_along(letters[1:10])
```

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

Generating Sequences: Caveats

- Be careful when writing `1:length(x)`.

```
means <- c()
# vector() function is useful for initiliazing empty vectors of known type
out <- vector(mode = "list", length = length(means))
for (i in 1:length(means)) {
    out[[i]] <- rnorm(10, means[[i]])
}
```

```
Error in rnorm(10, means[[i]]): invalid arguments
```

- As `:` works with both increasing and decreasing sequences:

```
1:length(means)
```

```
[1] 1 0
```

Generating Sequences: Caveats

- Use `seq_along(x)` instead.

```
1 seq_along(means)
```

```
integer(0)
```

```
1 means <- c()
2 # vector() function is useful for initiliazing empty vectors of known type
3 out <- vector(mode = "list", length = length(means))
4 for (i in seq_along(means)) {
5     out[[i]] <- rnorm(10, means[[i]])
6 }
```

```
1 out
```

```
list()
```

Iteration: `break` and `next`

- `break` - terminates the loop in which it is contained
- `next` - exits the iteration of a loop in which it is contained

```
for (i in seq(1, 6)) {
  if (i %% 2 == 0) {
    break
  }
  print(i)
}
```

```
[1] 1
```

```
for (i in seq(1, 6)) {
  if (i %% 2 == 0) {
    next
  }
  print(i)
}
```

```
[1] 1
[1] 3
[1] 5
```

Infinite Loops

Infinite Loops

- Loops that have no explicit limits for the number of iterations are called **infinite**.
- They have to be terminated with a `break` statement (or `Ctrl/Cmd-C` in interactive session).
- Such loops can be:
  - Unintentional (bug) or
  - Desired (e.g. waiting for user's input, some event).

```
i <- 1
while (TRUE) {
  i <- i + 1
  if (i > 10) {
    break
  }
}
```

```
i
```

```
[1] 11
```

Iteration: `repeat`

- `repeat` - defines code which is executed iteratively until the loop is explicitly terminated
- Is equivalent to `while (TRUE)`.

```
repeat {
  <some_code>
}
```

```
i <- 1
repeat {
  i <- i + 1
  if (i > 10) {
    break
  }
}
```

```
i
```

```
[1] 11
```

`for` vs `while` vs `repeat`

- Any loop written with `for` can be re-written with `while`.
- Any loop written with `while` can be re-written with `repeat`.
- I.e. in terms of flexibility: `repeat` > `while` > `for`
- However, it is a good idea to use the least flexible solution to a problem.

Next

- Tutorial: Implementing conditional statements and loops
- Assignment 1: Due at 12:00 on Monday, 6th October (submission on Blackboard)
- Next week: Functions in R