Week 3: Control Flow in R

POP77001 Computer Programming for Social Scientists

Tom Paskhalis

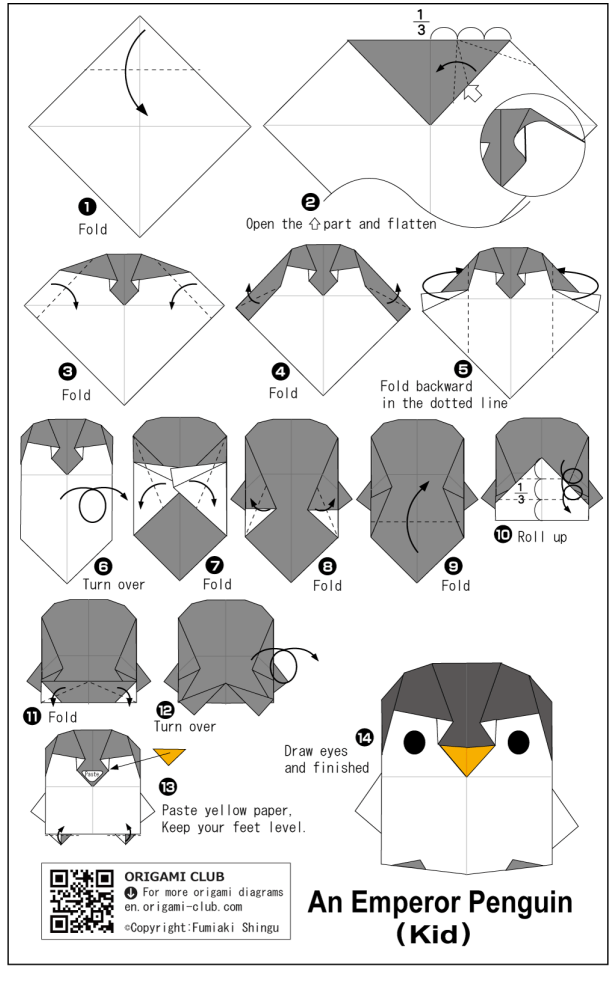
26 September 2022

Module website: tinyurl.com/POP77001

Overview

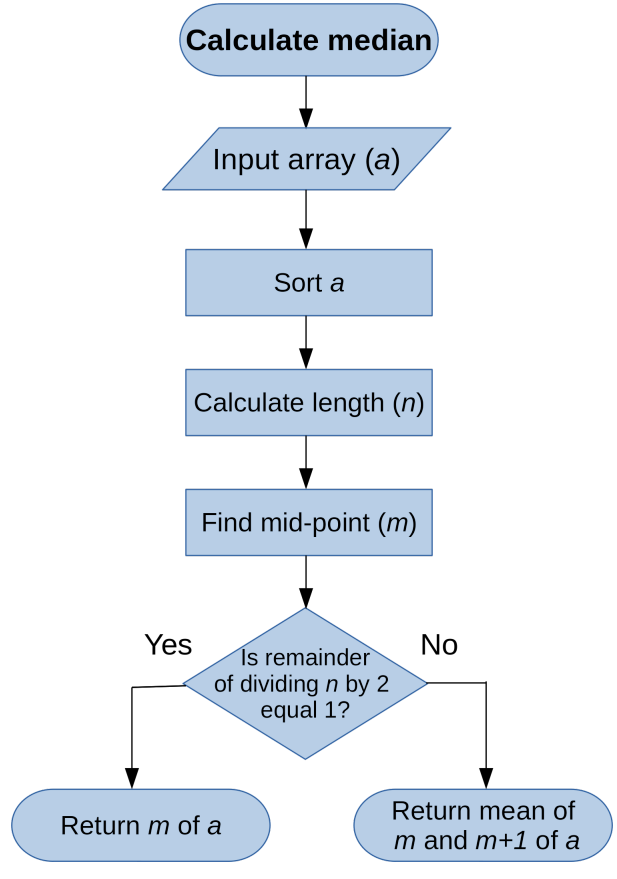
- Straight-line and branching programs
- Algorithms
- Conditional statements
- Loops and Iteration

Algorithm Example

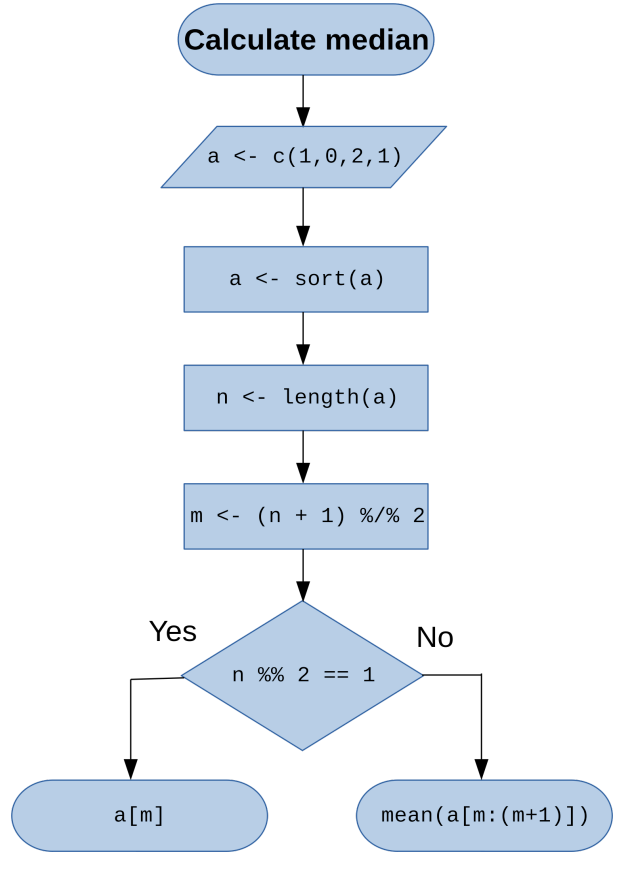


Source: Origami Club

Algorithm flowchart



Algorithm flowchart (R)



Calculate median

Calculate median

In [2]:
```
a <- c(1,0,2,1) # Input vector (1-dimensional array)
a <- sort(a) # Sort vector
a
```

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

Calculate median

In [2]:
```
a <- c(1,0,2,1) # Input vector (1-dimensional array)
a <- sort(a) # Sort vector
a
```

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

In [3]:
```
n <- length(a) # Calculate length of vector 'a'
n
```

```
[1] 4
```

Calculate median

In [2]:
```
a <- c(1,0,2,1) # Input vector (1-dimensional array)
a <- sort(a) # Sort vector
a
```

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

In [3]:
```
n <- length(a) # Calculate length of vector 'a'
n
```

```
[1] 4
```

In [4]:
```
m <- (n + 1) %/% 2 # Calculate mid-point, %/% is operator for integer d
m
```

```
[1] 2
```

Calculate median

In [2]:
```
a <- c(1,0,2,1) # Input vector (1-dimensional array)
a <- sort(a) # Sort vector
a
```

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

In [3]:
```
n <- length(a) # Calculate length of vector 'a'
n
```

```
[1] 4
```

In [4]:
```
m <- (n + 1) %/% 2 # Calculate mid-point, %/% is operator for integer d
m
```

```
[1] 2
```

In [5]:
```
n %% 2 == 1 # Check whether the number of elements is odd, %% (modulo)
```

```
[1] FALSE
```

Calculate median

In [2]:
```
a <- c(1,0,2,1) # Input vector (1-dimensional array)
a <- sort(a) # Sort vector
a
```

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

In [3]:
```
n <- length(a) # Calculate length of vector 'a'
n
```

```
[1] 4
```

In [4]:
```
m <- (n + 1) %/% 2 # Calculate mid-point, %/% is operator for integer d
m
```

```
[1] 2
```

In [5]:
```
n %% 2 == 1 # Check whether the number of elements is odd, %% (modulo)
```

```
[1] FALSE
```

In [6]:
```
mean(a[m:(m+1)])
```

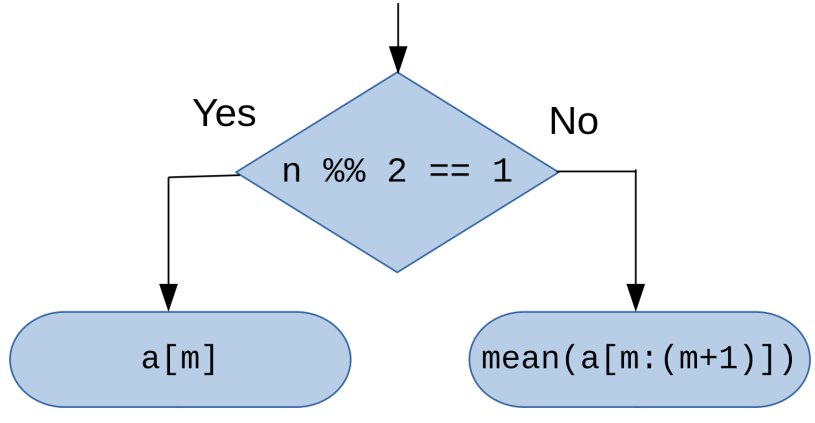
```
[1] 1
```

Control flow in R

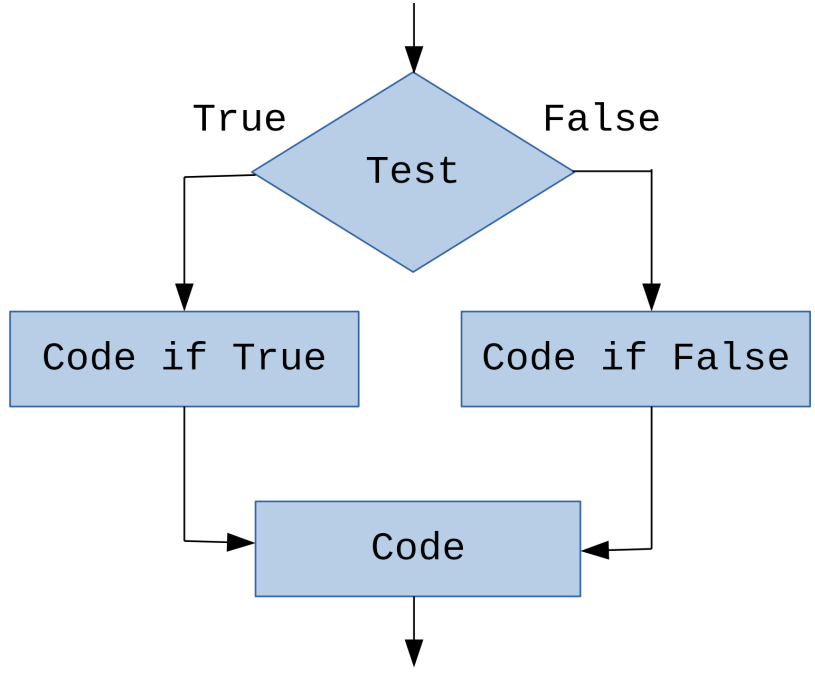
- *Control flow* is the order in which statements are executed or evaluated
- Main ways of control flow in R:
    - *Branching* (conditional) statements (e.g. `if`)
    - *Iteration* (*loops*) (e.g. `for`)
    - *Function calls* (e.g. `length()`)

Extra: R documentation on control flow

Branching programs



Conditional statements



Conditional statements: `if`

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

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

Conditional statements: `if`

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

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

In [7]:

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

```
[1] 1
```

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>
}
```

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>
}
```

In [8]:
```
a <- c(1, 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
```

Conditional statements: `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>
}
```

Example of longer conditional statement

Example of longer conditional statement

In [9]:
```
x <- 42
if (x > 0) {
  print("Positive")
} else if (x < 0) {
  print("Negative")
} else {
  print("Zero")
}
```

```
[1] "Positive"
```

Optimising conditional statements

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

Optimising conditional statements

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

In [10]:
```r
# Ask for user input and cast as double
num <- as.double(readline("Please, enter a number:"))
if (num %% 2 == 0) {
  print("Even")
} else if (num %% 2 == 1) {
  print("Odd")
} else {
  print("This is a real number")
}
```

```
Please, enter a number:43
[1] "Odd"
```

Nesting conditional statements

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

Nesting conditional statements

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

In [11]:
```
num <- as.integer(readline("Please, enter a number:")) # Ask for user i
if (num > 0) {
  if (num %% 2 == 0) {
    print("Positive even")
  } else {
    print("Positive odd")
  }
} else if (num < 0) {
  if (num %% 2 == 0) {
    print("Negative even") # Notice that odd/even check appears twice
  } else {
    print("Negative odd") # Consider abstracting this as a function
  }
} else {
  print("Zero")
}
```

```
Please, enter a number:-43
[1] "Negative odd"
```

`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

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

`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

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

In [12]:
```
num <- 1:10
num
```

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

`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

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

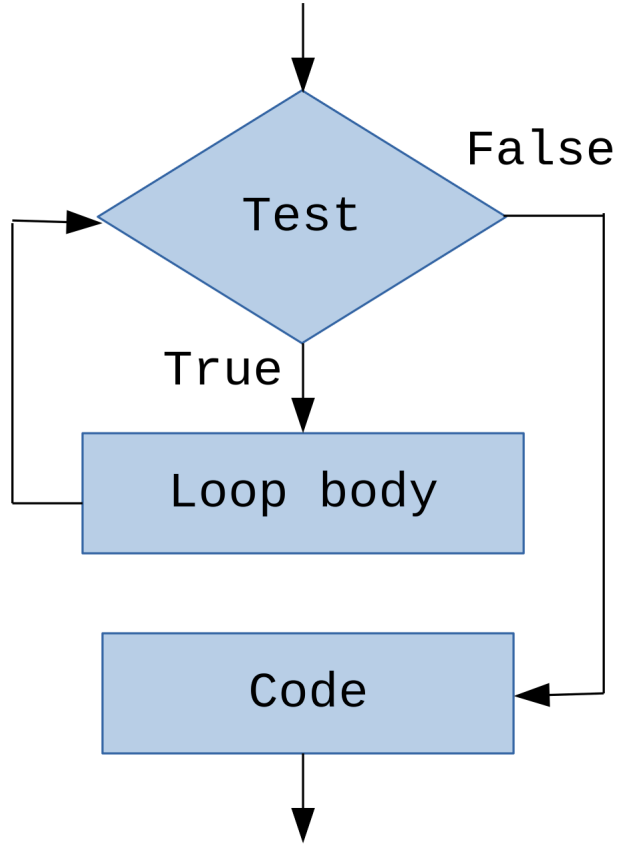
In [12]:
```
num <- 1:10
num
```

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

In [13]:
```
ifelse(num %% 2 == 0, "even", "odd")
```

```
 [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "o
dd"  "even"
```

Iteration (looping)



Iteration: `while`

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

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

Iteration: `while`

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

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

In [14]:
```
# 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
```

Iteration: `for`

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

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

Iteration: `for`

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

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

In [15]:
```
test <- c("t", "e", "s", "t")
for (i in test) {
  # cat() function concatenates and prints objects' representations
  cat(paste0(i, "!"), "")
}
```

```
t! e! s! t!
```

Iteration with conditional statements

Iteration with conditional statements

In [16]:
```r
# Find maximum value in a vector with exhaustive enumeration
v <- c(3, 27, 9, 42, 10, 2, 5)
max_val <- NA
for (i in v) {
  if (is.na(max_val) | i > max_val) {
    max_val <- i
  }
}
max_val
```

```
[1] 42
```

Generating sequences for iteration

- `seq()` function that we encountered in subsetting can be used in looping
- As well as its cousins: `seq_len()` and `seq_along()`

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

Generating sequences for iteration examples

Generating sequences for iteration examples

In [17]:
```
# If by argument is omitted, it defaults to 1
s <- seq(1, 20)
s
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 2
0
```

Generating sequences for iteration examples

In [17]:
```
# If by argument is omitted, it defaults to 1
s <- seq(1, 20)
s
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 2
0
```

In [18]:
```
# seq_len() is equivalent to seq(1, length(<object>))
seq_len(length(s))
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 2
0
```

Generating sequences for iteration examples

In [17]:
```
# If by argument is omitted, it defaults to 1
s <- seq(1, 20)
s
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 2
0
```

In [18]:
```
# seq_len() is equivalent to seq(1, length(<object>))
seq_len(length(s))
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 2
0
```

In [19]:
```
seq_along(s)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 2
0
```

Generating sequences for iteration examples

In [17]:
```
# If by argument is omitted, it defaults to 1
s <- seq(1, 20)
s
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 2
0
```

In [18]:
```
# seq_len() is equivalent to seq(1, length(<object>))
seq_len(length(s))
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 2
0
```

In [19]:
```
seq_along(s)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 2
0
```

In [20]:
```
# The sequence that you are supplying to seq_along() doesn't have to be
seq_along(letters[1:20])
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 2
```

Generating sequences for iteration examples continued

Generating sequences for iteration examples continued

In [21]:
```r
# vector() function is useful for initiliazing empty vectors of known t
s2 <- vector(mode = "double", length = length(s))
for (i in seq_len(length(s))) {
    s2[i] <- s[i] * 2
}
```

Generating sequences for iteration examples continued

In [21]:
```
# vector() function is useful for initiliazing empty vectors of known t
s2 <- vector(mode = "double", length = length(s))
for (i in seq_len(length(s))) {
    s2[i] <- s[i] * 2
}
```

In [22]:
```
s2
```

```
 [1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 4
0
```

Generating sequences for iteration examples continued

In [21]:
```r
# vector() function is useful for initiliazing empty vectors of known t
s2 <- vector(mode = "double", length = length(s))
for (i in seq_len(length(s))) {
    s2[i] <- s[i] * 2
}
```

In [22]:
```r
s2
```

```
 [1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 4
0
```

In [23]:
```r
s3 <- vector(mode = "double", length = length(s))
for (i in seq_along(s)) {
    s3[i] <- s[i] * 3
}
```

Generating sequences for iteration examples continued

In [21]:
```
# vector() function is useful for initiliazing empty vectors of known t
s2 <- vector(mode = "double", length = length(s))
for (i in seq_len(length(s))) {
    s2[i] <- s[i] * 2
}
```

In [22]:
```
s2
```

```
 [1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 4
0
```

In [23]:
```
s3 <- vector(mode = "double", length = length(s))
for (i in seq_along(s)) {
    s3[i] <- s[i] * 3
}
```

In [24]:
```
s3
```

```
 [1]  3  6  9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 6
0
```

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

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

In [25]:
```
for (i in seq(1,6)) {
  if (i %% 2 == 0) {
    break
  }
  print(i)
}
```

```
[1] 1
```

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

In [25]:
```
for (i in seq(1,6)) {
  if (i %% 2 == 0) {
    break
  }
  print(i)
}
```

```
[1] 1
```

In [26]:
```
for (i in seq(1,6)) {
  if (i %% 2 == 0) {
    next
  }
  print(i)
}
```

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

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)

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)

In [27]:
```
i <- 1
while (TRUE) {
  i <- i + 1
  if (i > 10) {
    break
  }
}
```

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)

In [27]:
```
i <- 1
while (TRUE) {
  i <- i + 1
  if (i > 10) {
    break
  }
}
```

In [28]:
```
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>
}
```

Iteration: `repeat`

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

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

In [29]:

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

Iteration: `repeat`

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

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

In [29]:
```
i <- 1
repeat {
  i <- i + 1
  if (i > 10) {
    break
  }
}
```

In [30]:
```
i
```

```
[1] 11
```

Infinite loop

```
while (TRUE)
```

Next

- Tutorial: Implementing conditional statements and loops
- Assignment 1: Due at 23:59 on Friday, 30th September (submission on Blackboard)
- Next week: Functions in R