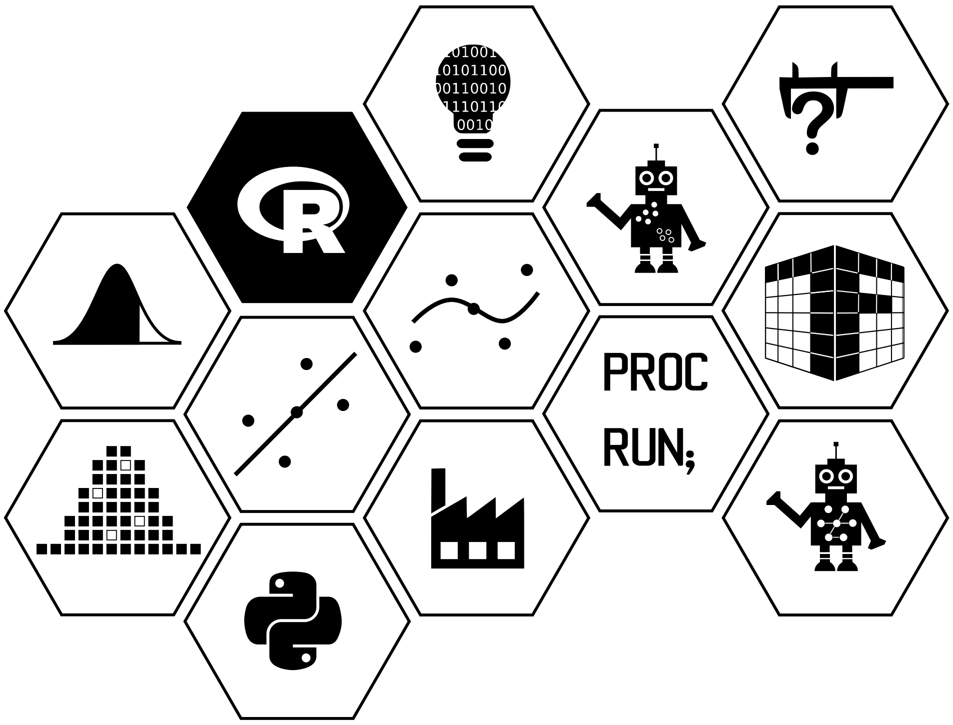
R Programming/ Statistical Computing

Craig Alexander

Academic Year 2022-23

Week 7:

Control Structures



DATA ANALYTICS GLASGOW

Logical switches: If statements

If statements

https://youtu.be/1Akkbt1aWxA

Duration: 7m14s

We have already made use of logical expressions when subsetting vectors or matrices:

```
x <- rnorm(10)                          # Generate ten realisations from N(0,1)
x[x<0] <- 0                             # Set all negative x to 0
```

This week we will learn control structures which let us perform such operations in a less cryptic way.

```
x <- rnorm(10)                          # Generate ten realisations from N(0,1)
x <- ifelse(x<0, 0, x)                  # Set all negative x to 0
```

Before covering the function `ifelse` we start with basic `if` statements. With `if` statements R can be programmed to take entirely different actions under different circumstances. `If` statements are in some way yes/no questions: depending on a condition R will either take one specified action or another one.

The basic form of an `if` statement is:

```
if (condition) {
    statement11
    ...
    statement1m
} else {
    statement21
    ...
    statement2n
}
```

`condition` is a logical expression, i.e. a scalar expression that evaluates to either `TRUE` or `FALSE`. If `condition` evaluates to `TRUE` R will run the statements in the first branch (`statement11` to `statement1m`). If however `condition` evaluates to `FALSE` then R will run the statements in the second branch (`statement21` to `statement2n`). The second `else`-part of the `if` statement is optional. If each branch only consists of a single statement the curly brackets can be omitted.

We can use the logical operators `!`, `&` `&&`, `|` and `||` as well an the functions such as `all`, `any` and `xor` in `condition` to combine logical expressions.

Note that the `condition` of an `if` statement *cannot* be a vector (of length > 1). If we want to carry out a conditional operation on a vector, we need to either subset it, use loops or the `ifelse` function.

It is common to indent the content of the two branches. The R interpreter ignores indentation, however properly indented code is much easier to read (for a human). Other programming languages enforce indentation, so it is good practice to get into this habit.

Example 1.

The if statement in

```
x <- 2
if (x==2) {
  print("x is 2")
} else {
  print("x is not 2")
}

## [1] "x is 2"
```

checks whether `x` is 2 and then prints `"x is 2`" on the screen.

Example 2.

In this example we will set `y` to $\sqrt{x}$ if x is non-negative. Otherwise we set it to $-\sqrt{-x}$.

```
x <- rnorm(1)
if (x>0) {
  y <- sqrt(x)
} else {
  y <- -sqrt(-x)
}
```

This is equivalent to setting `y` to `sign(x)*sqrt(abs(x))`.

Task 1.

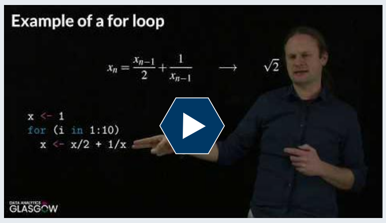
Create two variables `x` and `y` containing one random number each. Use an `if` statement to set the smaller of the two variables to the value of the larger variable.

Background reading: Sections 4.1.2 of A First Course in Statistical Programming with R

Section 4.1.2 covers `if` statements.

Loops

For loops


Example of a for loop
$x_n = \frac{x_{n-1}}{2} + \frac{1}{x_{n-1}} \longrightarrow \sqrt{2}$
x <- 1
for (i in 1:10)
x <- x/2 + 1/x
GLASGOW

For loops

https://youtu.be/BivoKnOakVQ

Duration: 8m38s

In the Week 2 material, we have looked at the so-called "Babylonian Method" for finding $\sqrt{2}$: the sequence defined by

$$x_n = \frac{x_{n-1}}{2} + \frac{1}{x_{n-1}}$$

tends to $\sqrt{2}$ as $n \longrightarrow \infty$ (provided $x_0 > 0$).

In order to approximate $\sqrt{2}$ using R we first set `x` to an initial value, say 1,

```
x <- 1
```

and then had to repeatedly (say 10 times) update `x` using the recursive formula from above.

```
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
x <- x/2 + 1/x
```

Loops are a way of doing these "repetitive" steps in a more elegant (and flexible) way:

```
x <- 1
for (i in 1:10)
    x <- x/2 + 1/x
```

The general syntax of `for` loops is

```
for (variable in sequence) {
    statement1
    ...
    statementn
}
```

The `for` loop executes the statements in the body of the loop (`statement1` to `statementn`) once for every element of `sequence`: the first time `variable` is set to the first element of `sequence` and the statements in the body are run using that value of `variable`, the second time `variable` is set to the second element of `sequence` and the statements in the body are run using `variable` set to that value, and so on.

If we wish to only iterate one statement we can omit the curly brackets.

Example 3.

A simple example illustrating a `for` loop is

```
for (i in 1:3)
    print(i)
```

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

Example 4.

The sequence we iterate over does not need to consist of numbers (though this is very often the case). We can use

```
for (day in c("M", "Tu", "W", "Th", "F"))
    print(day)
```

```
## [1] "M"
## [1] "Tu"
## [1] "W"
## [1] "Th"
## [1] "F"
```

Example 5.

We can use a `for` loop with an `if` statement to set all negative values of a vector `x` to 0.

```
x <- rnorm(10)
for (i in seq_along(x))
    if (x[i]<0)
        x[i] <- 0
x
```

```
##  [1] 0.0000000 0.0000000 0.9588504 0.0000000 0.0000000 0.0000000
##  [7] 2.3761935 0.1045770 0.0000000 0.5033510
```

In this loop we want to iterate over the length of the vector `x`. We could have used `1:length(x)`. However, this will not work if the length of `x` is 0. `1:length(x)` would then return the sequence $(1, 0)$, rather than a sequence of length 0 as would be required. The function `seq_along(x)` does exactly the same as `1:length(x)`, except that it handles the case of a vector of length zero correctly.

Example 6 (AR(1) process).

In this example we will use a `for` loop to generate a random sample of size 1000 from the model for an auto-correlated time series (you will learn more about time series in the Advanced Predictive Models course):

$$\begin{aligned} X_1 &\sim \mathrm{N}(0,1) \\ X_i|X_{i-1}=x_{i-1} &\sim \mathrm{N}(0.8x_{i-1}, 0.6^2) \end{aligned}$$

One can show that the second line is equivalent to setting $X_i = 0.8 \cdot X_{i-1} + 0.6 \cdot \epsilon_i$, where $\epsilon_i \sim \mathrm{N}(0,1)$

We start with creating an empty vector of the required size and setting its first entry to a random number drawn from the $\mathrm{N}(0,1)$ distribution:

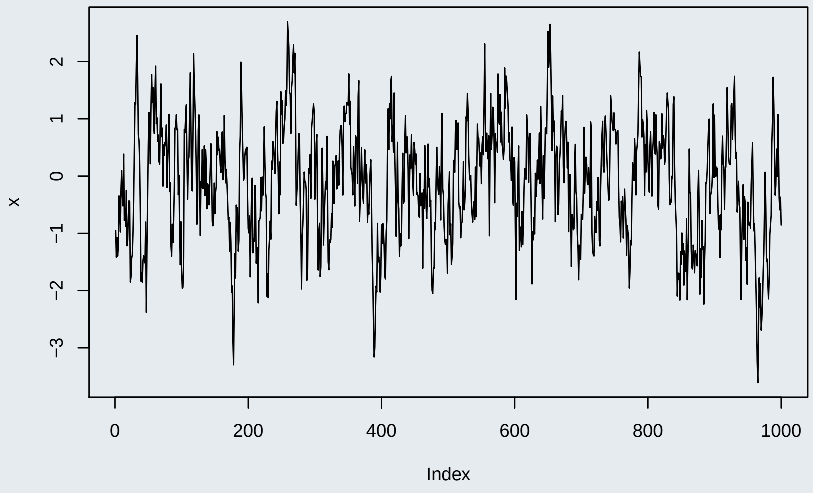
```
n <- 1000
x <- numeric(n)
x[1] <- rnorm(1)
```

We simulate all the remaining entries using a `for` loop:

```
for (i in 2:n) {                          # We start the loop at i=2!
    epsilon <- rnorm(1)
    x[i] <- 0.8*x[i-1] + 0.6 * epsilon
}
```

Finally, we can plot the results

```
plot(x, type="l")
```



Task 2.

Create a vector x containing some missing values using

```
x <- rnorm(100)                      # Generate white noise
x[sample(100,10)] <- NA              # Sneak in 10 missing values
```

Use a loop to create a vector `y` which contains the entries of `x`, however with missing values replaced by 0. (Remember, you can use the function `is.na` to test whether a value is missing).

Can you do the same without a loop?

Task 3.

In the setting of the previous task, suppose that rather than setting the missing values to 0 we want to omit them from `y`. We can do so by setting

```
y <- x[!is.na(x)]                    # Only copy non-missing entries
```

Can you modify your loop from task 2 to achieve the same?

Nested loops

Loops can be nested within each other. Note that you have to use different names for the loop variables in nested loops. In the examples below the outer loop uses `i`, whereas the inner loop uses `j`.

Example 7.

The following simple example illustrates how a nested loop works.

```
for (i in 1:2)
    for (j in 1:3)
        print(c(i,j))
```

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

The nested loop loops over all combinations of $i \in \{1, 2\}$ and $j \in \{1, 2, 3\}$. The index `j` changes fast, whereas the index `i` changes slowly.

Example 8.

When we looked at the standard plotting functions in R we created an image plot of the density of the bivariate normal distribution.

For the plotting function `persp` (or `filled.contour` or `image`) we need to store the values we want to plot in a matrix $\mathbf{Z} = (Z_{ij})$ with

$$Z_{ij} = \phi(x_i) \cdot \phi(y_j)$$

At the time, we used `expand.grid` to create all combinations of x_i and y_i in long format and then used `tidyr` to convert this into "wide" matrix format.

We could have also used a nested loop (which would however be slower). We start by creating the sequences `x` and `y` and then create an empty matrix to hold the Z_{ij}'s.

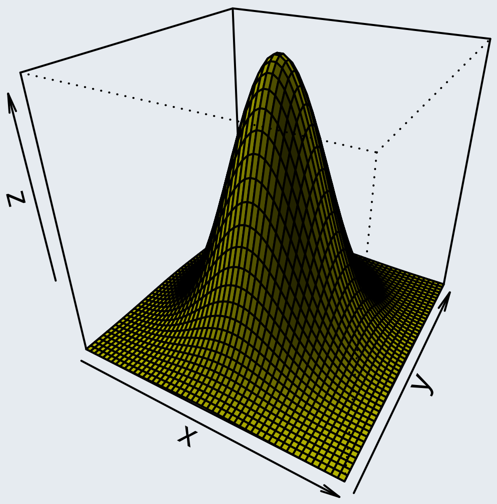
```
x <- seq(from=-3, to=3, length.out=50)      # Create sequence of grid for the x axis
y <- seq(from=-3, to=3, length.out=50)      # Create sequence of grid for the y axis
z <- matrix(nrow=length(x), ncol=length(y)) # Create matrix to store function values
```

In order to set every value of the matrix we need to go through all rows and all columns. Thus we need two loops nested within each other:

```
for (i in seq_along(x))                     # For all rows ...
    for (j in seq_along(y))                 # For all columns ...
        z[i,j] <- dnorm(x[i])*dnorm(y[j])   # Compute f(x,y)
```

Now we can create the plot:

```
persp(x, y, z, theta=30, phi=30, col="yellow", shade=0.5)
```



break and next

A `for`-loop repeats the statements in it a fixed number of times. The `break` statement gives additional flexibility and allows for aborting the loop immediately and before the sequence of indices has been finished. It is typically used inside an `if` statement.

Example 9.

In the motivating example at the start of this section we computed $\sqrt{2}$ using the Babylonian method, which is based on the iteration $x_n = \frac{x_{n-1}}{2} + \frac{1}{x_{n-1}}$. We implemented it using a `for` loop.

```
x <- 1
for (i in 1:10)
    x <- x/2 + 1/x
```

After the 10 iterations we obtained $x_{10} = 1.4142135623730949$, which is quite close to $\sqrt{2} = 1.4142135623730951$. Often, we are only interested in the first 8 digits, so we could have stopped the loop earlier, as soon as `x` changes by less than say 10^{-8}. This can be done using `break`.

In order to be able to quantify by how much `x` has changed we need to store the previous value of `x` in a variable `x.old`:

```
x <- 1
for (i in 1:100) {
    x.old <- x                               # Store the old value of x
    x <- x/2 + 1/x                           # Update x
    if (abs(x-x.old)<1e-8)                   # Check for convergence
        break
}
```

The `break` statement will abort the loop as soon as the change in `x` is less than 10^{-8}. To find out how many iterations were necessary, we can simply print `i` after running the loop:

```
i
```

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

Thus we needed only 5 iterations to obtain $\sqrt{2}$ to a precision of $\pm 10^{-8}$.

Task 4.

In example 9 we have used a variable `x` to store the current value of x in the recursive sequence. We had to introduce `x.old` to store the old value of `x`, so that we can compare to the current value in order to check for convergence.

Instead we could have used a vector `x` of length 100 and stored the value at the `i`-th iteration x_i in the `i`-th entry of `x`. After the end of the loop the required value is then in `x[i]`. Rewrite the loop in that way.

`next` halts the processing of the current iteration and goes back to the start of the body of the loop (using the next value of `sequence` in a `for` loop). `next` is the R equivalent of `continue` in C or Java.

Example 10.

The loop

```
for (x in 1:10) {
    if (x%%2==0)
        next
    print(x)
}
```

```
## [1] 1
## [1] 3
## [1] 5
## [1] 7
## [1] 9
```

prints all the odd numbers: if `x` is an even number `x%%2` is `0`, `next` is called, and thus the remainder of the statements in the body are skipped (before `x` is printed) and R continues with the next iteration.

In nested loops, `break` and `next` only affects the inner-most loops. R does not support `break`ing outer loops.

while loops

There are occasions when we need to repeatedly perform some operations, but we do not know in advance for how many times. As we have just seen, we can use `break` to stop the loop early. Another option is to use `while` loops (and `repeat` loops, which is the same as `while` loop without a condition).

The syntax of a `while` loop is

```
while (condition) {
    statement1
    ...
    statementn
}
```

The `while` loop checks `condition` each time before executing the first statement in the body of the loop. It executes the loop only if the `condition` evaluates to `TRUE`. As soon as the `condition` evaluates to `FALSE` for the first time, the loop is aborted.

Example 11.

We consider again the Babylonian method for finding $\sqrt{2}$. Just like in example 9, we want to stop iterating as soon as the change in `x` is small enough.

```
x <- 1
x.old <- 0
while(abs(x-x.old)>1e-8) {
    x.old <- x                              # Store the old value of x
    x <- x/2 + 1/x                          # Update x
}
x
```

```
## [1] 1.414214
```

Example 12 (A for loop using while).

We have seen that we can print the numbers 1 to 3 using a for loop:

```
for (i in 1:3)
    print(i)
```

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

We can do the same using a `while` loop, but we have to “manage” `i` ourselves.

```
i <- 1
while (i<=3) {
    print(i)
    i <- i+1
}
```

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

Blockly games

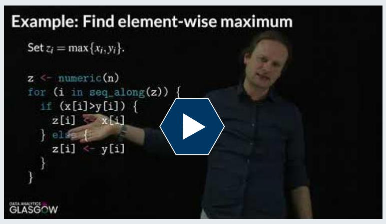
https://blockly-games.appspot.com/

Google has developed Blockly as a framework for creating interactive educational programming games. The idea behind Blockly games is that they visually illustrate control flow. Especially if you are new to programming, taking a look at some of these games might help develop your understanding of control structures and how they are best employed to solve a problem. The Blockly games generate JavaScript code, but being a “curly-bracket language” the control structures in JavaScript are very similar to the ones used by R (just ignore the semicolons at the end of each line).

Background reading: Sections 4.1.1 and 4.1.3 of A First Course in Statistical Programming with R

Sections 4.1.1 and 4.1.3 cover loops using `for` and `while`.

The function ifelse


Example: Find element-wise maximum

Set $z_i = \max\{x_i, y_i\}$.

```
z <- numeric(n)
for (i in seq_along(z)) {
  if (x[i]>y[i]) {
    z[i] <- x[i]
  } else {
    z[i] <- y[i]
  }
}
```

The function ifelse

https://youtu.be/aeDz66Jopeg

Duration: 6m33s

We have seen that the condition of an `if` statement has to be of length 1. If we want to use a condition that is of length greater then 1, we need to use a loop to go through the vector one-by-one. The function `ifelse` provides a more compact way of doing so. In some way, `ifelse` is the vectorised sibling of `if` statements.

The function

```
result <- ifelse(condition, yes, no)
```

sets the i-th element of the result to the i-th element of `yes` if the i-th element of `condition` is `TRUE`, otherwise it will be set to the i-th element of `no`.

The length of `result` will be the same as the length of `condition`. If `yes` and `no` are shorter than `condition`, `ifelse` will use the standard recycling rules.

Example 13.

We return to the example in which we set the negative entries of a vector `x` to 0.

If we want to use an `if` statement, we have to use a loop to go through the vector `x` one-by-one.

```
for (i in seq_along(x))
    if (x[i]<0)
        x[i] <- 0
```

Using `ifelse` simplifies this a lot:

```
x <- ifelse(x<0, 0, x)
```

Note that the assignment (`<-`) is always *outside* the call to `ifelse`, whereas that assignment is typically *inside* the `if` statement.

Example 14.

Suppose we have two vectors `x` and `y`

```
n <- 5
x <- sample(n)
y <- sample(n)
```

each containing the integers 1 to 5 in a random order.

Suppose you want to set the i-th entry of a new vector `z` to $z_i = \max\{x_i, y_i\}$. You can do this by placing an `if` statement inside a `for` loop.

```
z <- numeric(n)                 # Create vector to hold result
for (i in 1:n) {
  if (x[i]>y[i]) {              # If x[i] is larger ...
    z[i] <- x[i]                # ... set z[i] to  x[i]
  } else {                      # Otherwise (i.e. if x[i] is not larger) ...
    z[i] <- y[i]                # ... set z[i] to  y[i]
  }
}
```

It is a lot easier to use `ifelse`.

```
z <- ifelse(x>y, x, y)
```

Note that we could have also used subsetting.

```
z <- x                          # Start with a copy of x
select <- y>x                   # Find out for which entries y is larger than x
z[select] <- y[select]          # Set these to y
```

Task 5.

Without running the code in R, determine what `ifelse` returns in the code snippet below.

```
x <- c(1,2,9)
y <- c(2,6,4)
z <- c(3,5,7)
ifelse(x<4, y, z)
```

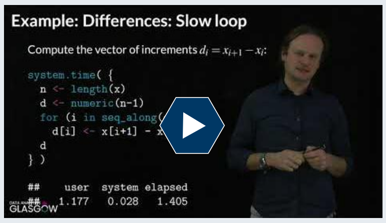
Task 6.

What does the following loop do?

```
x <- rnorm(10)                        # Generate some white noise
out <- numeric(length(x))
for (i in seq_along(x)) {
    if (x[i]>0) {
        out[i] <- x[i]
    } else {
        out[i] <- -x[i]
    }
}
```

Rewrite the code so that it uses the function `ifelse`.

Avoiding loops



Example: Differences: Slow loop

Compute the vector of increments $d_i = x_{i+1} - x_i$:

```
system.time( {
  n <- length(x)
  d <- numeric(n-1)
  for (i in seq_along(
    d[i] <- x[i+1] - x
  d
} )

##    user  system elapsed
##   1.177   0.028   1.405
```

Avoiding loops

https://youtu.be/R1rwOCUUebA

Duration: 10m32s

Loops are relatively slow in R. Code usually runs faster and can become more legible when avoiding loops. R's vectorised nature makes this particularly easy.

Example 15 (The sum of two vectors).

Suppose we want to calculate the sum of two vectors x and y of length 100,000.

```
n <- 1e5
x <- rnorm(n)
y <- rnorm(n)
```

The easiest and fastest way is to exploit that R can add vectors together using the operator +.

```
system.time(z <- x + y)
```

```
##    user  system elapsed
##   0.000   0.000   0.001
```

Using a loop to set the entries z one-by-one is a lot slower:

```
system.time( {
  z <- numeric(n)                  # Create vector of correct size
  for (i in 1:n)                   # Set entries one-by-one
    z[i] <- x[i]+y[i]
} )
```

```
##    user  system elapsed
##   0.070   0.005   0.075
```

An even less efficient approach would consist of creating a vector z of (initially) zero length, and then appending the newly computed z_i one by one.

```
system.time( {
  z <- c()
  for (i in 1:n)
    z <- c(z, x[i]+y[i])
} )
```

```
##    user  system elapsed
##  10.268   0.069  10.355
```

This is awfully slow. The reason why this approach is so slow is that in every iteration z is replaced by a new vector. Memory for the new vector needs to be allocated, the current vector z needs to be copied into the new vector z, and finally the old vector z needs to be deleted from the memory.

Our code would be equally slow if we were sloppy when initialising the vector z and create a vector of zero length. This is valid as R increases the size of the vector as needed, but brings with it the same issues of having to repeatedly copy the vector as it is being extended.

```
system.time( {
  z <- c()                           # Create an empty vector and let R extend it
  for (i in 1:n)                     # Set entries one-by-one
      z[i] <- x[i]+y[i]
})
```

```
##    user  system elapsed
##   0.075   0.001   0.076
```

We will now look at a less straightforward example showcasing how using vector-based operations and subsetting can speed up code (and yield more compact code).

Example 16 (Increments).

Suppose we compute the vector of increments $d_i = x_{i+1} - x_i$. Our first approach uses a loop.

```
system.time( {
    n <- length(x)
    d <- numeric(n-1)
    for (i in 1:(n-1))
        d[i] <- x[i+1] - x[i]
} )
```

```
##    user  system elapsed
##   0.072   0.000   0.072
```

We cannot simply set `d` to the difference of `x` and `x`, as we subtract x_i from x_{i+1}. Essentially we need to offset the two copies of `x` before we subtract them. We can do this using

```
system.time( {
    n <- length(x)
    d <- x[-1] - x[-n]
} )
```

```
##    user  system elapsed
##   0.001   0.000   0.000
```

which is a lot faster. (The previous video explains this trick in more detail).

Answers to tasks

Answer to Task 1. You can use the following R code.

```
x <- rnorm(1)
y <- rnorm(1)
if (x<y) {
    x <- y
} else {
    y <- x
}
```

The code is equivalent to setting

```
x <- y <- max(x,y)
```

Answer to Task 2. You can use the following loop

```
y <- numeric(length(x))
for (i in seq_along(x))
    if (!is.na(x[i])) {
        y[i] <- x[i]
    } else {
        y[i] <- 0
    }
```

In this example it is a lot easier to use subsetting.

```
y <- x
y[is.na(y)] <- 0
```

Answer to Task 3. We create an empty vector `y` and only append the non-missing entries from `x`.

```
y <- c()
for (i in seq_along(x))
    if (!is.na(x[i]))
        y <- c(y, x[i])
```

The loop is a lot slower than the subsetting approach and also less clear to read (for a human).

Answer to Task 4. We can store the entire sequence (rather than just the store the current value) using the following R code.

```
n <- 100
x <- numeric(n)
x[1] <-  1
for (i in 2:100) {
    x[i] <- x[i-1]/2 + 1/x[i-1]           # Update x
    if (abs(x[i]-x[i-1])<1e-8)            # Check for convergence
        break
}
x[i]
```

```
## [1] 1.414214
```

Answer to Task 5. The call to `ifelse` returns the vector $(2, 6, 7)$. The condition evaluates to (TRUE,TRUE,FALSE), so the result is set to $(y_1, y_2, z_3) = (2, 6, 7)$.

Answer to Task 6. The loop stores the modulus ("absolute value") of `x` in `out`. This can be recoded using `ifelse` as

```
out <- ifelse(x>0, x, -x)
```

We could, of course, also have simply set

```
out <- abs(x)
```