You may have seen the memes going around about fun ways to program the straightforward function `isEven()` which returns `TRUE` if the input is even, and `FALSE` otherwise. I had a play with this and it turned into enough for a blog post, and a nice walk through some features of R.

The 'traditional' way to check if an integer is even is to check if it is divisible by 2. This can be achieved with the modulo operator `%%` which gives the remainder after dividing by another number. For example, `5 modulo 2` or `5 %% 2` gives 1 because 2 goes into 5 twice with 1 leftover. If a number `x` is even, it is an exact multiple of 2, and so `x %% 2 == 0`.

```
5 %% 2
```

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

```
6 %% 2
```

```
## [1] 0
```

A function which tests values of `x` for this property could be written as

```
## 1
isEven <- function(x) {
    ## traditional modulo check
    x %% 2 == 0
}
```

The `==` operation checks that the left side is equal to the right side (but not necessarily identical, e.g. the classes can be different) and returns either `TRUE` or `FALSE` (or `NA`, but that's not an issue for the cases we're looking at here). I've also relied on the fact that the result of the last statement in a function body is used as the return value if no explicit `return()` is used.

Confirming that this works is as easy as trying some values. It's always good to check that your function produces results you expect. It's also a good idea to try some odd values to ensure you don't hit edge-cases.

```
isEven(0)
```

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

```
isEven(3)
```

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

```
isEven(4)
```

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

```
isEven(-1)
```

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

```
isEven(-6)
```

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

In the process of playing with this function I refined how I tested my code. I started with a set of confirming evaluations like above. Then I wanted to confirm that they actually gave the results I expect, so I refined it to

```
test_isEven <- function() {
    all(
        isEven(0) == TRUE,
        isEven(3) == FALSE,
        isEven(4) == TRUE,
        isEven(-1) == FALSE,
        isEven(-6) == TRUE
    )
}
```

Now I just had one function to call which confirmed that all these tests gave the expected results. Once I had explained this layout to myself with the word 'expected', I realised what I actually wanted was a test suite, and `testthat` is a great candidate for that. Refactoring the above into a series of expectations might look like

```
library(testthat)
test_isEven <- function() {
    test_that("isEven peforms as expected", {
        expect_true(isEven(0))
        expect_false(isEven(3))
        expect_true(isEven(4))
        expect_false(isEven(-1))
        expect_true(isEven(-6))
    })
}
```

Now I can test any implementation of `isEven()` with just one function call, and if one of the expectations fails I'll know which it is. Running this with the above `isEven()` produces no output, so the tests succeeded

```
test_isEven()
```

The 'no output' might be concerning, so we can also run a negative control to make sure it breaks when things are broken. Let's break the `isEven()` by reversing the test

```
## 1a
isEven <- function(x) {
    ## (broken) traditional modulo check
    x %% 2 == 1
}
test_isEven()
```

```
## Error: Test failed: 'isEven peforms as expected'
## * :4: isEven(0) isn't true.
## * :5: isEven(3) isn't false.
## * :6: isEven(4) isn't true.
## * :7: isEven(-1) isn't false.
## * :8: isEven(-6) isn't true.
```

So, we can trust that if we make a mistake or don't implement this properly, we'll know. Typically a function you write would have a lot more safety checking, such as ensuring that we actually passed a value, and that it's an integer, but for the sake of this post I'm going to assume that these are both guaranteed to be true.

This version of `isEven()` is simple and it works, but that's not what the internet wants – a common challenge is to make a version of `isEven()` which *doesn't* use modulo. Now we need to think a little more, but we can at least check any implementation with our tests.

I came up with a few, both from borrowing from other solutions and on my own. Let's see…

If the last digit is any of 0, 2, 4, 6, or 8, then it's an even number

```
## 2
isEven <- function(x) {
    ## ends with an even digit
    grepl("[02468]$", x)
}
test_isEven()
```

With that same idea, if the least significant bit (binary) is unset then it's even

```
## 3
isEven <- function(x) {
    ## least significant bit is unset
    x == 0 || !bitwAnd(x, 1)
}
test_isEven()
```

Continuing down the bitwise path, if we can shift left and right and get back to the original number, then it's even

```
## 4
isEven <- function(x) {
    ## bitwise shift right then left
    !(x-(bitwShiftL(bitwShiftR(x, 1), 1)))
}
test_isEven()
```

If we alternate `FALSE` and `TRUE` counting from 0 to `x` then we get our answer

```
## 5
isEven <- function(x) {
    ## alternate TRUE/FALSE
    y <- FALSE
    for (i in 0:x) {
        y <- !y
    }
    return(y)
}
test_isEven()
```

We could do the same thing with recursion

```
## 6
isEven <- function(x) {
    ## recursion, n-1 is odd if n is even
    if (x == 0) return(TRUE)
    !isEven(abs(x) - 1)
}
test_isEven()
```

Not quite using modulo, integer division by 2, doubled, should return the original value

```
## 7
isEven <- function(x) {
    ## integer division, doubled
    2*(x %/% 2) == x
}
```

```
test_isEven()
```

Similarly, the result of regular division cast to integer, doubled, should return the original value

```
## 7a
isEven <- function(x) {
    ## normal division, doubled
    2*as.integer(x/2) == x
}
test_isEven()
```

If we start from a number and count towards 0 by twos then we will hit 0 if the number is even

```
## 8
isEven <- function(x) {
    ## moving by 2s towards 0 ends at 0
    y <- x
    repeat({
        if (y == 0) return(TRUE)
        if (sign(x) != sign(y)) return(FALSE)
        y <- y - sign(x)*2
    })
}
test_isEven()
```

We can write that a bit simpler if we only use the absolute value of `x`

```
## 8a (abs version)
isEven <- function(x) {
    ## moving by 2s towards 0 ends at 0
    y <- abs(x)
    repeat({
        if (y == 0) return(TRUE)
        if (y < 0) return(FALSE)
        y <- y - 2
    })
}
test_isEven()
```

Exploiting mathematical properties, we know that -1 to any even power returns 1

```
## 9
isEven <- function(x) {
    ## -1 to an even power is 1
    (-1)**x == 1
}
test_isEven()
```

The relationship \[\cos(2x) = -\cos(x)\] can also be exploited

```
## 10
isEven <- function(x) {
    ## cos(2x) == -cos(x)
    cos(x*pi) == -cos(pi)
}
test_isEven()
```

Now for some more R-specific solutions… R rounds towards even integers, and we can exploit that

```
## 11
isEven <- function(x) {
    ## R rounds even real numbers down
    round(x + 0.5) == x
}
test_isEven()
```

We can create a vector of ‘every other integer’ and check whether a value is in there

```
## 12
isEven <- function(x) {
    ## is x in set of 'every other integer'?
    abs(x) %in% (0:abs(x))[c(TRUE, FALSE)]
}
test_isEven()
```

Creating a vector of `TRUE` and `FALSE` we can extract the element corresponding to a value

```
## 13
isEven <- function(x) {
    ## even/odd sequence
    if (x == 0) return(TRUE)
    rep(c(FALSE, TRUE), (abs(x)/2) + 1)[abs(x)]
}
test_isEven()
```

Then, starting to get really absurd, we could solve the equation \[2n = x\] which will have an integer `n` if `x` is even

```
## 14
isEven <- function(x) {
    ## integer solution to 2n = x?
    n <- solve(2, x)
    as.integer(n) == n
}
test_isEven()
```

And, lastly, for the truly absurd, we can use the fact that “zero” and “eight” are the only single digits written as English words with an “e”. This requires a couple of extra packages, but can be done.

```
## 15
isEven <- function(x) {
    ## zero and eight are the only odd
    ## last digit as words with an e
    last <- english::words(as.integer(stringr::str_sub(x, -1, -1)))
    last == "zero" || last == "eight" || !grepl("e", last)
}
test_isEven()
```

This isn’t an exhaustive list, but it seemed like a good place to stop looking. If you can think of more then add them to this thread on Twitter.

I hope this demonstrates the usefulness of writing functions and testing them with `testthat`. Plus, if the `%%` operator ever breaks, you have plenty of alternatives.

```
devtools::session_info()
```

```
## ─ Session info ─────────────────────────────────────────────────────────────
```

```
──
##  setting  value
##  version  R version 3.6.2 (2019-12-12)
##  os       Pop!_OS 19.04
##  system   x86_64, linux-gnu
##  ui       X11
##  language en_AU:en
##  collate  en_AU.UTF-8
##  ctype    en_AU.UTF-8
##  tz       Australia/Adelaide
##  date     2020-03-09
##
## ─ Packages ──────────────────────────────────────────────────────────────
──────
##  package     * version date       lib source
##  assertthat    0.2.1   2019-03-21 [1] CRAN (R 3.6.2)
##  backports     1.1.5   2019-10-02 [1] CRAN (R 3.6.2)
##  blogdown      0.18    2020-03-04 [1] CRAN (R 3.6.2)
##  bookdown      0.17    2020-01-11 [1] CRAN (R 3.6.2)
##  callr         3.4.2   2020-02-12 [1] CRAN (R 3.6.2)
##  cli           2.0.1   2020-01-08 [1] CRAN (R 3.6.2)
##  crayon        1.3.4   2017-09-16 [1] CRAN (R 3.6.2)
##  desc          1.2.0   2018-05-01 [1] CRAN (R 3.6.2)
##  devtools      2.2.2   2020-02-17 [1] CRAN (R 3.6.2)
##  digest        0.6.24  2020-02-12 [1] CRAN (R 3.6.2)
##  ellipsis      0.3.0   2019-09-20 [1] CRAN (R 3.6.2)
##  english       1.2-5   2020-01-26 [1] CRAN (R 3.6.2)
##  evaluate      0.14    2019-05-28 [1] CRAN (R 3.6.2)
##  fansi         0.4.1   2020-01-08 [1] CRAN (R 3.6.2)
##  fs            1.3.1   2019-05-06 [1] CRAN (R 3.6.2)
##  glue          1.3.1   2019-03-12 [1] CRAN (R 3.6.2)
##  htmltools     0.4.0   2019-10-04 [1] CRAN (R 3.6.2)
##  knitr         1.28    2020-02-06 [1] CRAN (R 3.6.2)
##  magrittr      1.5     2014-11-22 [1] CRAN (R 3.6.2)
##  memoise       1.1.0   2017-04-21 [1] CRAN (R 3.6.2)
##  pkgbuild      1.0.6   2019-10-09 [1] CRAN (R 3.6.2)
##  pkgload       1.0.2   2018-10-29 [1] CRAN (R 3.6.2)
##  prettyunits   1.1.1   2020-01-24 [1] CRAN (R 3.6.2)
##  processx      3.4.2   2020-02-09 [1] CRAN (R 3.6.2)
##  ps            1.3.2   2020-02-13 [1] CRAN (R 3.6.2)
##  R6            2.4.1   2019-11-12 [1] CRAN (R 3.6.2)
##  Rcpp          1.0.3   2019-11-08 [1] CRAN (R 3.6.2)
##  remotes       2.1.1   2020-02-15 [1] CRAN (R 3.6.2)
##  rlang         0.4.5   2020-03-01 [1] CRAN (R 3.6.2)
##  rmarkdown     2.1     2020-01-20 [1] CRAN (R 3.6.2)
##  rprojroot     1.3-2   2018-01-03 [1] CRAN (R 3.6.2)
##  sessioninfo   1.1.1   2018-11-05 [1] CRAN (R 3.6.2)
##  stringi       1.4.5   2020-01-11 [1] CRAN (R 3.6.2)
##  stringr       1.4.0   2019-02-10 [1] CRAN (R 3.6.2)
##  testthat    * 2.3.1   2019-12-01 [1] CRAN (R 3.6.2)
##  usethis       1.5.1   2019-07-04 [1] CRAN (R 3.6.2)
##  withr         2.1.2   2018-03-15 [1] CRAN (R 3.6.2)
##  xfun          0.12    2020-01-13 [1] CRAN (R 3.6.2)
##  yaml          2.2.1   2020-02-01 [1] CRAN (R 3.6.2)
##
## [1] /home/jono/R/x86_64-pc-linux-gnu-library/3.6
## [2] /usr/local/lib/R/site-library
```

```
## [3] /usr/lib/R/site-library
## [4] /usr/lib/R/library
```