



## 5. Over and over

Ken Rice  
Tim Thornton

University of Washington

*Seattle, July 2020*

## In this session

---

In Sessions 1–4, we completed tasks by breaking them down, into one line of an R script at a time. In principle, we could do *everything* this way. But;

- Repeating the same job many times (i.e. once for each person/guinea pig in the dataset) the typing gets slow & tedious, and is error prone
- For iterative methods, we don't know how much code will be needed before starting the task

This session, and the next, introduce writing loops, so we can re-use the same code in a script, without re-typing it.

NB This module does *not* cover every R tool for looping.

# A very first for() loop

---

Many people's first computer program looks like this;

```
> for(i in 1:5){  
+   print("hello world!")  
+   print(i^2)  
+ }  
[1] "hello world!"  
[1] 1  
[1] "hello world!"  
[1] 4  
[1] "hello world!"  
[1] 9  
[1] "hello world!"  
[1] 16  
[1] "hello world!"  
[1] 25
```

Two fundamental ideas;

- Go round the loop 5 times
- Each time, do something that may (or may not) depend on which 'go round' it is

Of course, for() loops also have more practical uses...

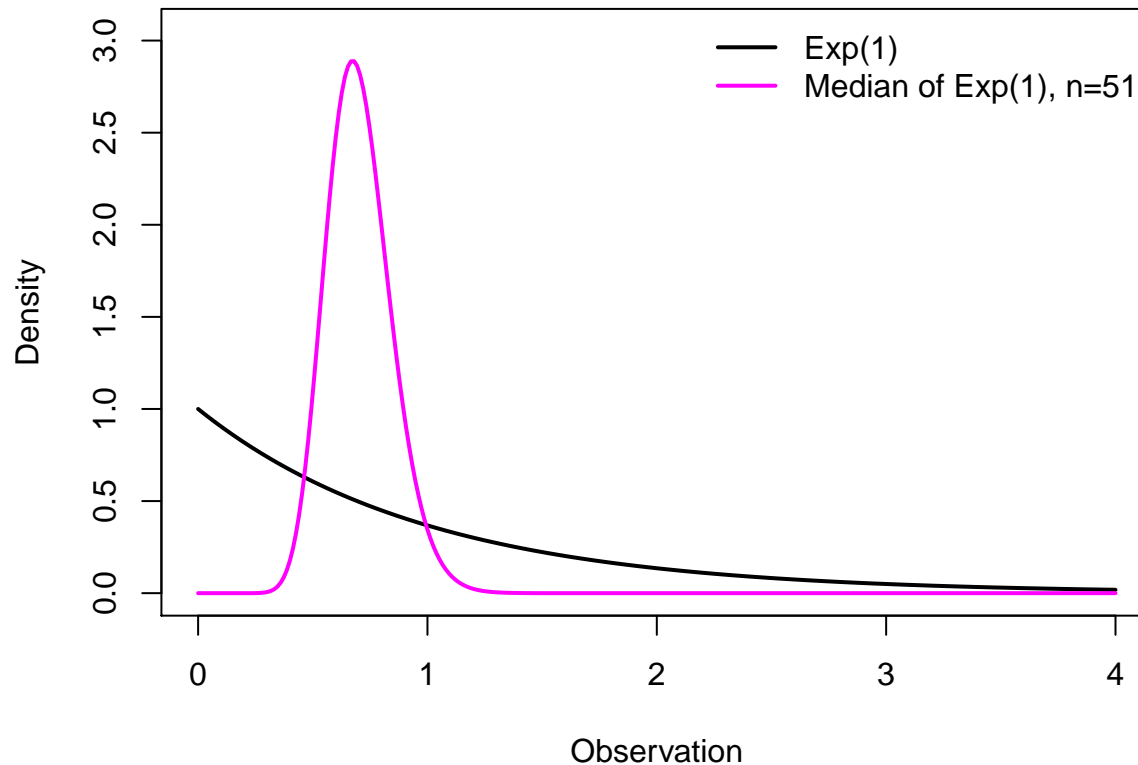
# Example: hard math made easy

---

A question from analysis of survival traits – and its answer!

*What is the expected value of the median of a sample, size  $n = 51$ , of independent data from  $Exp(1)$ ?*

*What is its variance?*



## Example: hard math made easy

---

If the picture didn't make it obvious enough (!) here are the *exact* answers;

$$\mathbb{E}[\text{Median}_{51}] = \frac{2178178936539108674153}{3099044504245996706400}$$

$$\mathbb{E}[\text{Median}_{51}^2] = \frac{2467282316063667967459233232139257976801959}{4802038419648657749001278815379823900480000}$$

These are 0.70286 and 0.51380 to 5 d.p. – so the variance is  $0.51380 - 0.70286^2 = 0.01978$ .

- Yes, there are ‘pretty’ answers here
- In general there aren't – but the ‘expectation’ ( $\mathbb{E}[\dots]$ ) terms just mean averaging over lots of datasets – which is easy, with a computer
- We can get a good-enough answer very quickly

# Example: hard math made easy

---

We'll write code that;

1. Generates a *single* sample of size  $n = 51$  from  $Exp(1)$
2. Calculates its median and stores this number
3. Repeats steps 1 and 2 many times, then works out the mean and variance of the stored numbers

Here are steps 1 and 2 – run them and see what's created;

```
many.medians <- vector(10000, mode="numeric") # or just rep(NA, 10000)
set.seed(4)
for(i in 1:10000){
  mysample <- rexp(n=51, rate=1)          # take a single sample, size 51
  many.medians[i] <- median(mysample)    # calculate & store its median
}
```

The function `set.seed()` tells R where to start its random-number generator – this is important, as it means we can repeat the code and get the same answers. Choose any 'seed' you like.

# Example: hard math made easy

---

How to think of the seed;



- The seed indicates starting place in the list
- The list closely *resembles* truly random numbers – certainly closely enough for our purposes – but is *actually* fixed

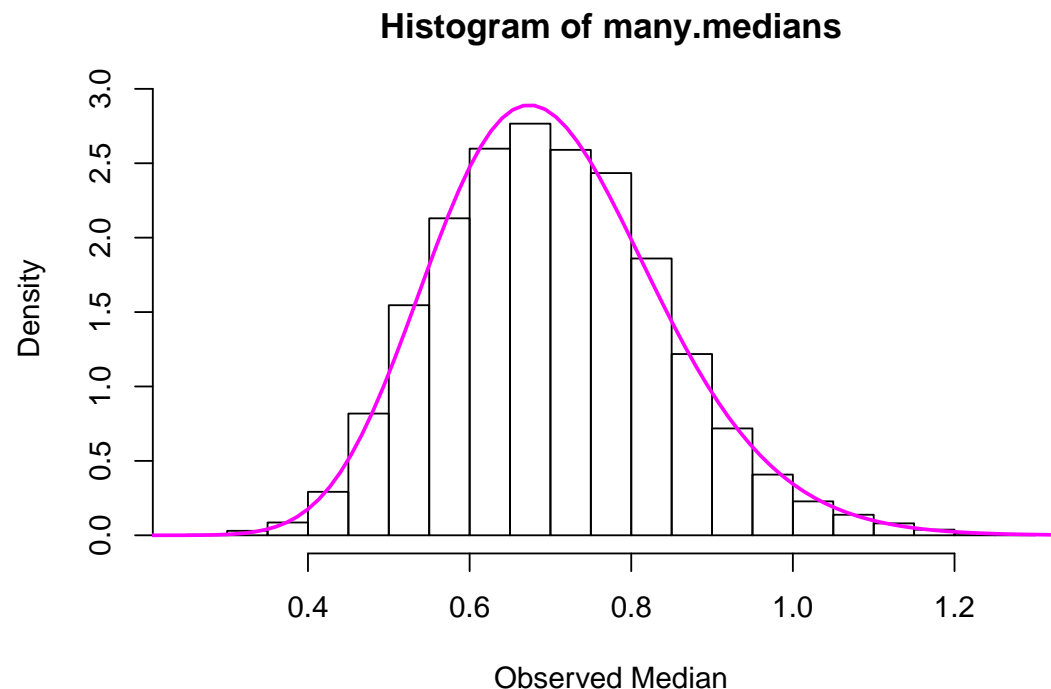
# Example: hard math made easy

---

And the answers, from 10,000 simulations, with that seed?

```
> mean(many.medians)
[1] 0.702171          # exact answer is 0.70286
> var(many.medians)
[1] 0.01955728       # exact answer is 0.01978
```

NB: for large-enough values of 10,000, we could work basically *anything* about the sample median, with little extra work;





## Example: hard math made easy

---

Notes on the coding; (NB see `?Control` for the help page on `for()`, `?for` won't work as `for` is 'restricted')

- `for( [iteration] in [vector of iteration values] )` – the vector of iteration values can be of anything, not just `1:n`
- The expression between the curly brackets `{ }` is evaluated each 'go round' the loop, substituting `i` for `1, 2, ... 10,000` in turn
- Very important – create an object to store the output first (but no need to create `i` first). To do this, you'll need to know how big the output is going to be.
- Last-used version of objects used (`i`, `mysample`) are available when the loop terminates – which is very helpful, if (when!) an error occurs
- We used `rexp()`, but there are *many* built-in distributions; `rnorm()`, `rgamma()`, `rbinom()`, `rpois()` etc

## Example: data manipulation

---

Recall the salary example – on faculty measured over several years. Suppose we were interested in the *final* observation for each person – how to construct that dataset?

- Different numbers of observations per person – so can't just look at e.g. rows 1,5,11,15, ... (but see `seq()` if you *do* want to do this)
- Different entry and exit years
- `subset()` won't work, neither will use of square brackets

Instead, we can go through every `id` number, pull out the rows with that `id` and record the one for which `year` is highest. Or, if the data is sorted first (by `id` and `time`) pull out the *last* row for each `id` number.

As before, it's very important that we prepare an object for the results ('pulled out' data, here) *before* running any loops.

## Example: data manipulation

---

First sort the data, and make the empty object ready to take output;

```
salary <- salary[ order(salary$id, salary$year), ]  
View(salary) # check we know what we should get from subsetting  
  
n <- length(unique(salary$id)) # how many individual people?  
finalsalary <- salary[0,] # take just column names from salary  
finalsalary[1:n,] <- NA    # fill in with missing values  
str(finalsalary)           # check the structure we made
```

- `order()` returns the vector that puts objects in order. There is a `sort()` function, but it accepts only vectors and not data frames
- A less-sneaky way to make a new empty data frame uses e.g. `data.frame(id=NULL, age=NULL, sex=NULL)`
- In RStudio, `View()` operates in the Source window; in vanilla R it opens up a new window. Neither refreshes automatically

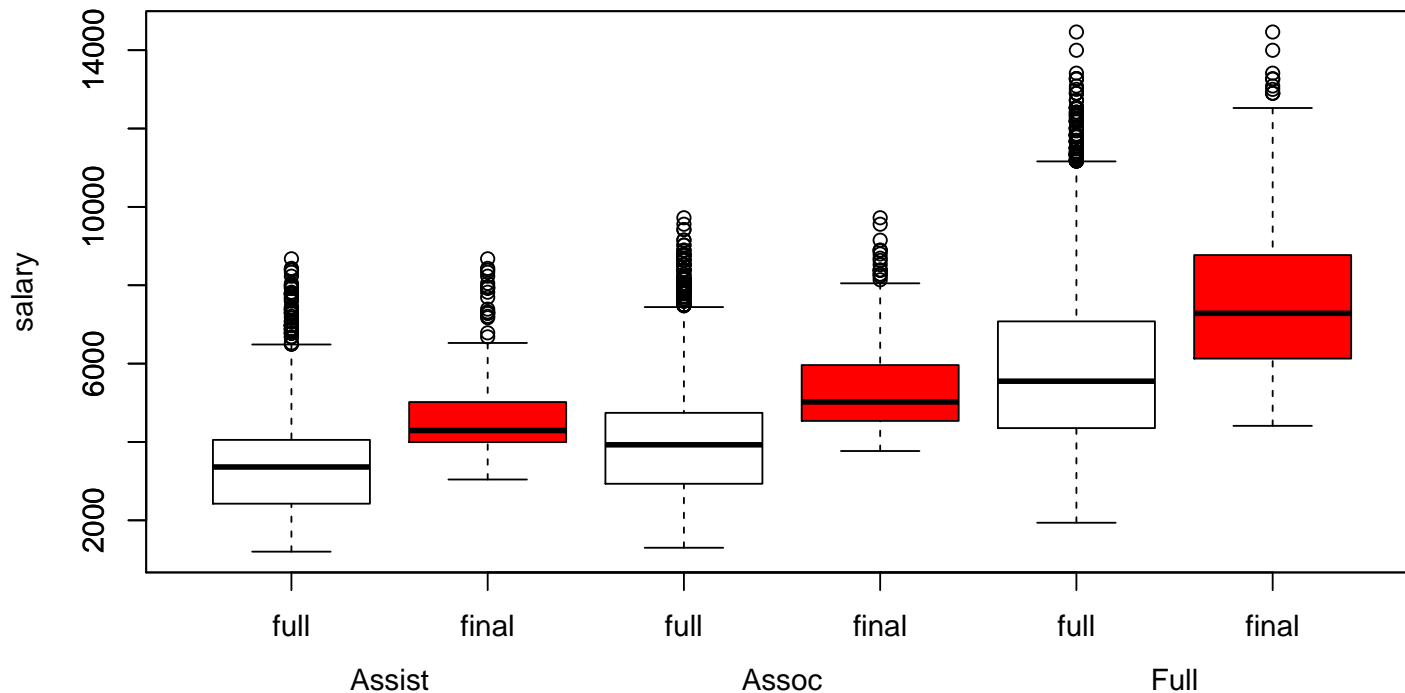
# Example: data manipulation

---

Now for the loop;

```
for(i in 1:n){  
  id.i      <- unique(salary$id)[i]  
  salary.i  <- subset(salary, id==id.i)  
  n.i       <- dim(salary.i)[1]      # dim() for dimension  
  finalsalary[i,] <- salary.i[n.i,]  # i.e. just the last row  
} # View(finalsalary) a good idea, to check it worked
```

Compare the full dataset (white) and final-only version (red);

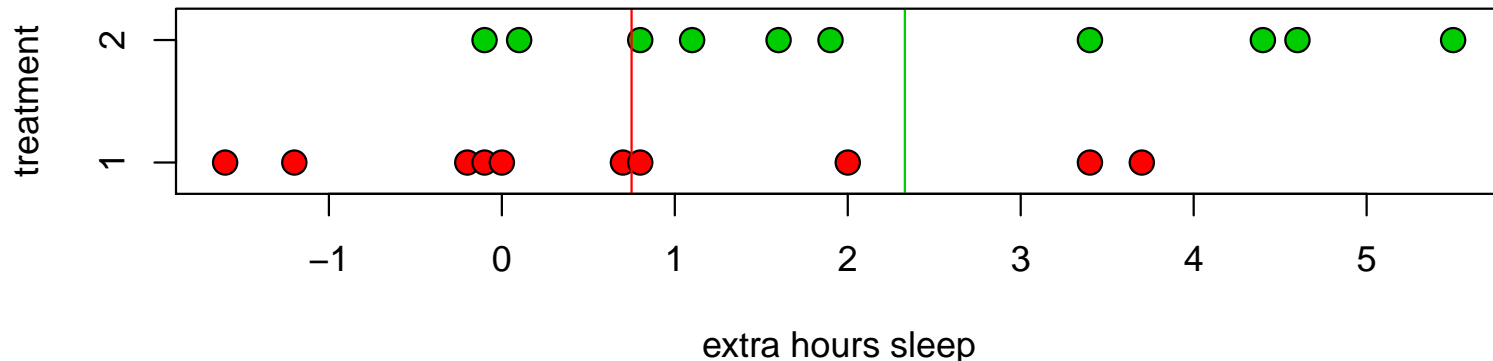


# Example: permutation test

---

A classical statistical question: are the data we've observed *unexpected*, if there's nothing going on?

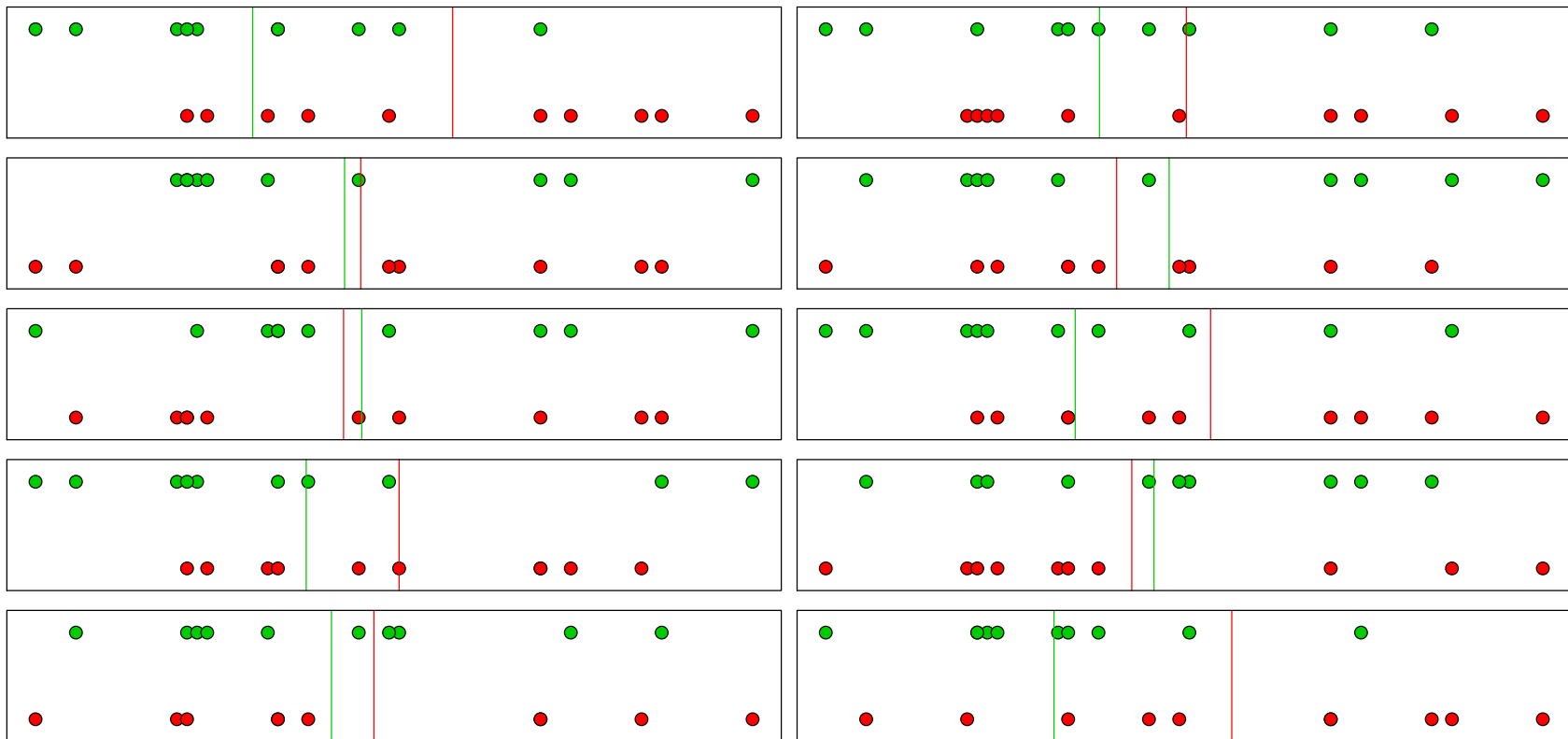
An example where we can answer this is R's `sleep` data;



- 10 subjects per group
- Groups receive different treatments, we record how many hours sleep they get, compared to baseline
- Mean extra hours sleep is higher in group 2 (2.33 hrs vs 0.75 hrs, so difference is 1.58 hrs)

# Example: permutation test

What if there were nothing going on\*, i.e. what if any differences in mean were just chance? If so, the data we saw would be just as likely as that obtained assigning the group labels *at random*;



\* Formally, what if the *null hypothesis* of equal means held, in the population from which this data has been sampled?

## Example: permutation test

---

To measure how unexpected our data is, we compute the red/green difference in means for many of these *permutations*, and see how the *observed* data compares.

```
orig.mean.diff <- with(sleep,
  mean(extra[group==2]) - mean(extra[group==1])
)
orig.mean.diff

many.mean.diff <- vector(10000, mode="numeric") # set up place to put output
set.seed(4)                                     # set the random seed

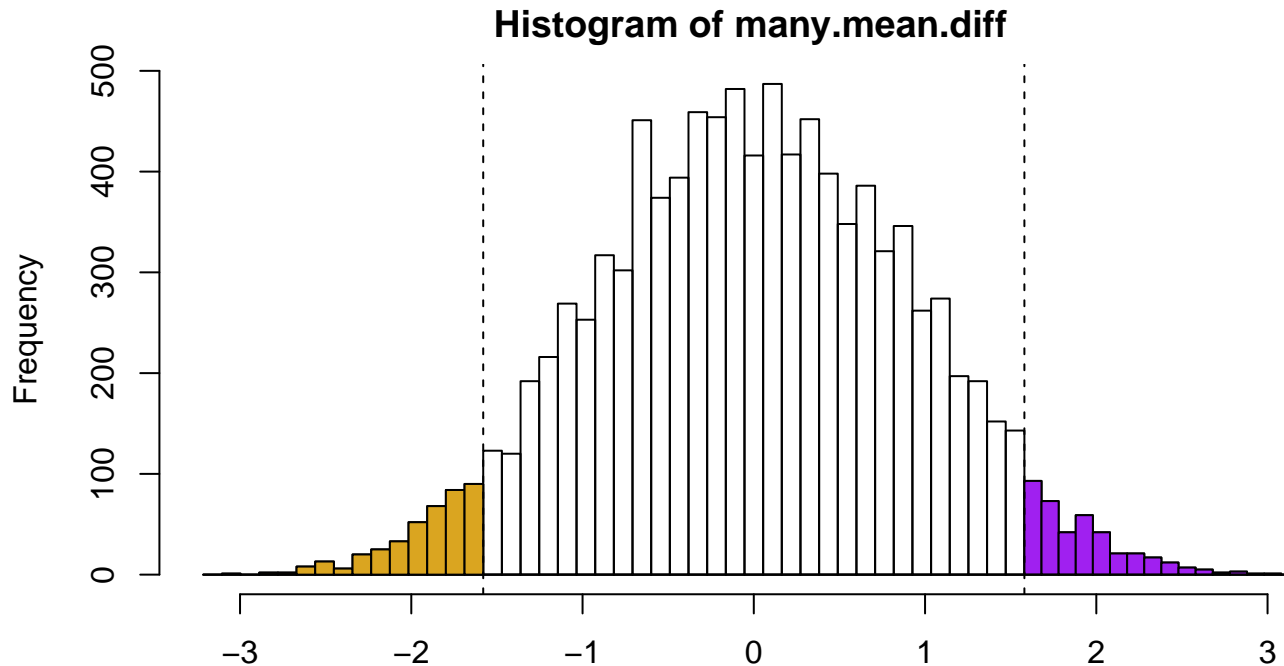
for(i in 1:10000){                             # do this bit 10,000 times
  group.shuffle <- sample(sleep$group)
  many.mean.diff[i] <- with(sleep,
    mean(extra[group.shuffle==2]) - mean(extra[group.shuffle==1])
  )
}
```

- `sample()` returns a random shuffle of a vector
- The same calculation is made, for the original data and the shuffled version; the difference in means is called the *test statistic*

# Example: permutation test

---

How does original data (w/ mean diff=1.58) compare to these?



```
> table(many.mean.diff>orig.mean.diff)
FALSE  TRUE
 9601   399
> mean(many.mean.diff>orig.mean.diff)
[1] 0.0399
> mean(abs(many.mean.diff)>abs(orig.mean.diff))
[1] 0.0789
```



## Example: permutation test

---

- The proportion of sample in the RH tail is a (valid)  $p$ -value for a one-tailed test, where the alternative is that green > red.  $p = 0.04$ , here
- The proportion in both tails is the  $p$ -value for a two-tail test;  $p = 0.079$
- There is some 'Monte Carlo' error in these  $p$ -values; roughly  $\pm 0.004$  here, i.e. 2 decimal places in  $p$ . If that's not good enough, use more permutations. (Here, could use all 184,756 – but in larger samples it's not possible)

To get a quicker (but approximate) version of the same thing;

```
> t.test(extra~group, data=sleep) # recall extra 'depends on' group
Welch Two Sample t-test
data:  extra by group
t = -1.8608, df = 17.776, p-value = 0.07939
alternative hypothesis: true difference in means is not equal to 0
```

The  $t$  test makes fewer assumptions than most people think!

# Notes on timing, and speed

---

Doing a lot of calculations can take a long time – it's useful to know how long. Try out the `system.time()` command on a smaller version of the problem, i.e.

```
system.time({  
  for(i in 1:1000){                                # just 1000, not 10000  
    group.shuffle <- sample(sleep$group)  
    many.mean.diff[i] <- with(sleep,  
      mean(extra[group.shuffle==2]) - mean(extra[group.shuffle==1])  
    )  
  }  
})
```

This returns the time taken to run the outer curly brackets;

```
user  system elapsed  
0.57   0.00   0.60
```

... so running 100,000 permutations would take  $100 \times 0.6 / 60 = 1$  minute, roughly. (NB this is much less time than it took to write the code!)

If RStudio hangs, there is a 'STOP' button on the Console window; in vanilla R hit Escape, or Ctrl-D.

# Notes on timing, and speed

---

Throughout, we have stressed the importance of setting up empty objects for the loop's output. Why? Let's code the permutation test without doing this;

```
many.mean.diff <- NULL          # this will 'grow', in the loop
system.time({
  for(i in 1:100000){
    group.shuffle <- sample(sleep$group)
    mean.diff <- with(sleep,
      mean(extra[group.shuffle==2]) - mean(extra[group.shuffle==1]))
    many.mean.diff <- c(many.mean.diff, mean.diff) # 'grow' the dataset
  })
```

```
   user  system elapsed # CPU/child process/total
115.53    4.93   122.07
```

- This works, but at half the speed of the other version
- The extra time is *all* spend copying vector `many.mean.diff` – R copies objects slowly
- The slowdown is worse for larger objects, i.e. gets worse with more permutations, i.e. when speed really matters

# Notes on timing, and speed

---

Compared to using a single R command (when available) to do the job, `for()` loops can be inefficient.

- Add two vectors (`x <- y + z`) don't add them element by element (`for(i in 1:n){ x[i] <- y[i] + z[i]}`)
- Recall `ifelse()` earlier, rather than looping over a vector.

```
many.samples <- matrix(data=NA, nrow=100000,ncol=20)
```

```
for(i in 1:100000){  
  many.samples[i,] <- sample(sleep$extra)  
}
```

```
many.mean.diff <- rowMeans(many.samples[,1:10]) - rowMeans(many.samples[,11:20])
```

- Shuffling the outcomes is equivalent to shuffling the groups
- A `matrix` has all entries of the same type – less flexible than a `data.frame`, but faster to work with
- This version takes 4.3s, i.e. it's  $\times 14$  faster than the loop.
- Not available for every task – also uses more memory

# Summary

---

- Writing loops saves a lot of typing – essential for serious computing jobs, but helpful for data management too
- `for()` loops offer enough flexibility for several jobs – more to come in the next session!
- As with all programming; break the job into lots of small pieces, and do each one in turn
- Never never *never* grow the output except when doing tiny jobs where speed is irrelevant, and then **only** if you promise not to fall into bad habits!
- Other looping methods exist in R – but aren't in this module