A case against pipes in R and what to do instead

Pipes (`%>%`) are great for improving readibility of lengthy data processing scripts, but I'm beggining to learn they have some weaknesses when it comes to large and complex data processing.

We are running a number of projects at the moment that require managing and wrangling large and complex datasets. We have numerous scripts we use to document our workflow and the data wrangling steps. This has turned out to be very helpful, because when we identify bugs in the end product, we can go back and fix them.

But I'm starting to see a pattern. Most of the really insidious bugs occur in sections of code that use `dplyr` tools and pipes. These are always the types of bugs that don't throw an error, so you get a result, it just turns out to be wrong. They are the worst kind of bugs. And hard to detect and fix.

So we are now moving away from using pipes in complex scripts. For simple scripts I intend to keep using them, they are so fast and easy. Here's what we're trying instead.

The problem with pipes

So here's some made up data that mimics the kind of fish survey data we often have:

```
sites <- data.frame(site = letters[1:5],
                    temp = rnorm(5, 25, 2), stringsAsFactors = FALSE)
dat <- expand.grid(site = letters[1:5],
                   transect = 1:4)
dat$abundance <- rpois(20, 11)
```

So we have site level data with a covariate, `temp` and transect level data with fish counts.

Now say we have an error and one of our sites has capitals, instead of lower case, so lets introduce that bug:

```
sites$site[1] <- "A"
```

Now if I join and summarize them, I will lose one of the sites

```
library(dplyr)

## Warning: package 'dplyr' was built under R version 3.6.3

##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

```
dat %>%
  inner_join(sites) %>%
  group_by(site) %>%
  summarize(mean(abundance))

## Joining, by = "site"

## Warning: Column `site` joining factor and character vector, coercing into
## character vector

## # A tibble: 4 x 2
##   site  `mean(abundance)`
##
## 1 b                    9
## 2 c                   12
## 3 d                   12.5
## 4 e                   12.8
```

Obvious enough here, but issues like that are much harder to detect in very large datasets.

Unit testing

The solution of course is to code in 'unit tests' to make sure each operations are doing what you expect. For small data you can just look, but for big datasets its not so easy.

For long pipes with multiple steps we'd usually do this debugging and testing interactively. So I'd write the first line (the join) save the output to a new variable, check it worked ok, then move on to write the next step of the pipe.

Now here's the catch. In complex project its common to change the data that goes into your pipe (in this case `dat` or `sites` dataframes). For instance, in our current project new data comes in all the time.

New data presents new issues. So a pipe that worked the first time may no longer work the second time.

This is why it is crucial to have unit tests **built into your code**.

There are lots of sophisticated R packages for unit testing, including ones that work with pipes. But given many of us are just learning tools like `dplyr` its not wise to add extra tools. So here I'll show some simple unit tests with base R.

Unit testing an example

Joins often case problems, due to mis-matching (e.g. if site names are spelt differently in different datasets, which is a very common human data entry error!).

So its wise to check the join has worked. Here's some examples:

```
dat2 <- inner_join(dat, sites)

## Joining, by = "site"

## Warning: Column `site` joining factor and character vector, coercing into
## character vector
```

Now compare number of rows:

```
nrow(dat2)
```

```
## [1] 16
```

```
nrow(dat)
```

```
## [1] 20
```

Obviously the join has lost data in this case.

We can do better though with a complex script. We'd like to have an error if the data length changes. We can do this:

```
nrow(dat2) == nrow(dat)
```

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

Which tells us TRUE/FALSE if the condition is met. To get an error use `stopifnot`

```
stopifnot(nrow(dat2) == nrow(dat))
```

Common unit tests for data wrangling

Of the top of my head here are a few of my most commonly used unit tests To check the number of sites has stayed the same, use `length(unique(`… to get the number of unique cases:

```
length(unique(dat$site))
```

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

```
length(unique(dat2$site))
```

```
## [1] 4
```

```
length(unique(dat$site)) == length(unique(dat2$site))
```

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

Or if we wanted to compare the `site` and `dat` dataframes:

```
unique(sites$site) %in% unique(dat$site)
```

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

The `%in%` just means are the sites names in `sites` matching the site names in `dat`? (We can use `stopifnot` here too, with multiple TRUE/FALSE values).

How many don't match?

```
sum(!(unique(sites$site) %in% unique(dat$site)))
```

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

The `!` is a logical 'not' (not FALSE = TRUE, so we are counting non-matches).

Which one doesn't match?

```
sites$site[!unique(sites$site) %in% unique(dat$site)]
```

```
## [1] "A"
```

Here's another insidious bug caused by joins, when our covariate dataframe has duplicate site entries:

```
sites <- data.frame(site = c(letters[1:5], "a"),
                    temp = c(rnorm(5, 25, 2), 11), stringsAsFactors = FALSE)
sites
```

```
##   site     temp
## 1    a 19.30061
## 2    b 23.76530
## 3    c 24.89018
## 4    d 25.16386
## 5    e 23.83092
## 6    a 11.00000
```

Now we have two sites called `a` with different values of `temp`. Check out the join:

```
dat2 <- inner_join(dat, sites)
```

```
## Joining, by = "site"
```

```
## Warning: Column `site` joining factor and character vector, coercing into
## character vector
```

```
nrow(dat)
```

```
## [1] 20
```

```
nrow(dat2)
```

```
## [1] 24
```

So its added rows, ie made up data we didn't have. Why? Well the join duplicated all the site `a` values for both values of temp:

```
filter(dat2, site == "a")
```

```
##   site transect abundance     temp
## 1    a        1        10 19.30061
## 2    a        1        10 11.00000
## 3    a        2        11 19.30061
## 4    a        2        11 11.00000
## 5    a        3        12 19.30061
## 6    a        3        12 11.00000
## 7    a        4         9 19.30061
## 8    a        4         9 11.00000
```

No watch this, we can really go wrong when we summarize:

```
dat2 %>%
  group_by(site) %>%
  summarize(sum(abundance))
```

```
## # A tibble: 5 x 2
##   site  `sum(abundance)`
##
## 1 a                   84
## 2 b                   36
## 3 c                   48
## 4 d                   50
## 5 e                   51
```

It looks like site `a` has twice as many fish as it really does (78, when it should have 39). So imagine you had a `site` dataframe you were happy worked, then your collaborator sent you a new one to use, but it had duplicate rows. If you didn't have the unit test to check your join in place, you may never know about this doubling of data error.

We could check for this by checking for the number of transects e.g.

```
dat_ntrans <- dat2 %>% group_by(site) %>% summarize(n = n())
dat_ntrans
```

```
## # A tibble: 5 x 2
##   site      n
##
## 1 a         8
## 2 b         4
## 3 c         4
## 4 d         4
## 5 e         4
```

```
dat_ntrans$n != 4
```

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

(Yes I used a pipe this time, but a simple one).