Pure, predictable, pipeable: creating fluent interfaces with R

Hadley Wickham
@hadleywickham
Chief Scientist, RStudio

January 2015

MOAH PIPES!

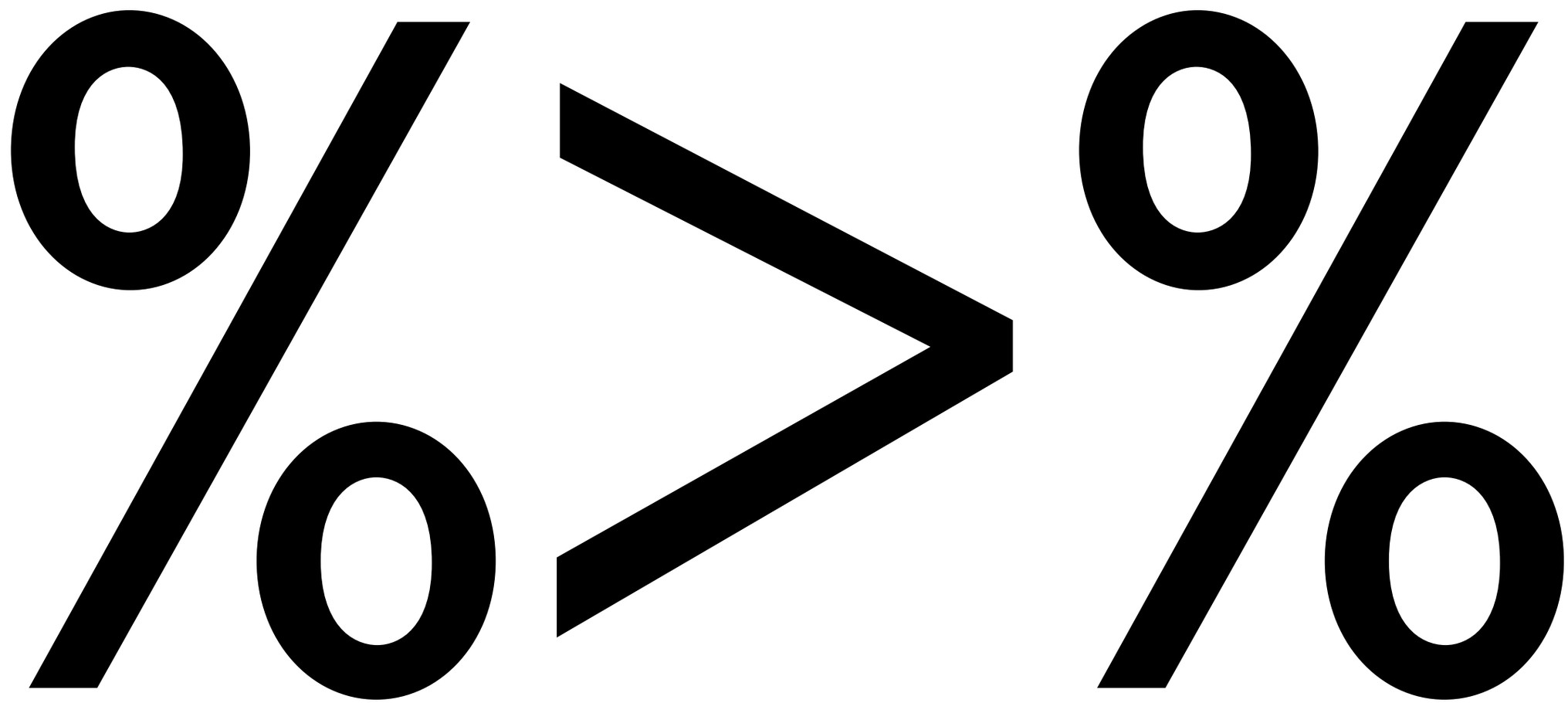
Hadley Wickham

@hadleywickham

Chief Scientist, RStudio

January 2015

magrittr::



```
x %>% f(y)
# f(x, y)

x %>% f(z, .)
# f(z, x)
```

```
x %>% f(y) %>% g(z)
# g(f(x, y), z)

# Turns function composition (hard to read)
# into sequence (easy to read)
```

```
foo_foo <- little_bunny()

foo_foo %>%
  hop_through(forest) %>%
  scoop_up(field_mouse) %>%
  bop_on(head)

# vs
bop_on(
  scoop_up(
    hop_through(foo_foo, forest),
    field_mouse
  ),
  head
)
```

```
# From http://zevross.com/blog/2015/01/13/a-new-data-processing-workflow-for-r-dplyr-magrittr-tidyr-ggplot2/

library(dplyr)
library(tidyr)

word_count <- shakespeare %>%
  group_by(word) %>%
  summarize(count=n(), total = sum(word_count)) %>%
  arrange(desc(total))

top8 <- shakespeare %>%
  semi_join(head(word_count, 8)) %>%
  select(-corpus_date) %>%
  spread(word, word_count, fill = 0)
```

```r
# Pipes for web scraping (hadley)
library(rvest)
lego_movie <- html("http://www.imdb.com/title/tt1490017/")

rating <- lego_movie %>%
  html_nodes("strong span") %>%
  html_text() %>%
  as.numeric()
```

```r
cast <- lego_movie %>%
  html_nodes("#titleCast .itemprop span") %>%
  html_text()

poster <- lego_movie %>%
  html_nodes("#img_primary img") %>%
  html_attr("src")
```

```r
# Functional programming pipes (hadley + lionelgit)
library(lowliner)

mtcars %>%
  split(.$cyl) %>%
  map(~ lm(mpg ~ wt, data = .)) %>%
  map(summary) %>%
  map_v("r.squared")
```

```
# Control a digitalocean machine (sckott + hadley)
library(analogsea)

droplet_create("my-droplet") %>%
  droplet_power_off() %>%
  droplet_snapshot() %>%
  droplet_power_on() %>%
```

```
# Create, modify & delete gists (sckott)
library(gistr)

gists("minepublic") %>%
  .[[1]] %>%
  add_files("~/alm_othersources.md") %>%
  update()

# http://recology.info/2015/01/gistr-github-gists/
```

```
# Ensure objects are of correct type (smbache)
library(ensurer)
the_matrix <-
  get_matrix() %>%
  ensure_that(is.numeric(.),
              NCOL(.) == NROW(.))
```

Goal: Solve **complex** problems by *combining* **simple** pieces.

http://brickartist.com/gallery/pc-magazine-computer/. CC-BY-NC

Principles

- **Pure**: each function is easy to understand in isolation.
- **Predictable**: once you've understood one, you've understood them all.
- **Pipeable**: combine simple pieces with a standard tool (%>%).

Pure

Goal: each function can be easily understood in isolation

A function is pure if:

(a) Its **output** only depends on its **inputs**

(b) It makes **no changes** to the state of the world

1 minute: what common R functions are impure?

```
# Lots of important functions are impure:

# Outputs don't depend only on inputs
runif(10)
read.csv()
Sys.time()

# Make changes to the world
library()
write.csv()
plot()
options()
source()
reference classes
environments
```

Why?

- Easier to reason about because you can understand them in isolation
- Trivial to parallelise
- Trivial to memoise (cache)

How?

- There are a lot of useful things you can't do with purity
- But you usually can isolate impurity to a handful of functions
- Doing so leads to code that's easier to understand and easier to repurpose
- Case study: `plot.lm()` vs. `fortify.lm()`

```
fortify.lm <- function(model, data = model$model, ...) {
  infl <- influence(model, do.coef = FALSE)
  data$.hat <- infl$hat
  data$.sigma <- infl$sigma
  data$.cooksd <- cooks.distance(model, infl)
  data$.fitted <- predict(model)
  data$.resid <- resid(model)
  data$.stdresid <- rstandard(model, infl)

  data
}
```

See also https://github.com/dgrtwo/broom

Predictable

Goal: once you've mastered one member of a class you've mastered them all

```
#rstats #wat


c(1, 2, 3)
c("a", "b", "c")
c(factor("a"), factor("b"))


diag(4:1)
diag(4:2)
diag(4:3)
diag(4:4)


nchar("NA")
nchar(NA)
```

```
# But more problematic
grepl(pattern, x)
gsub(pattern, replacement, x)
gregexpr(pattern, replacement, x)
strsplit(x, pattern)

# CSVs
read.csv(file)
write.csv(x, file)

# Tabular data
read.table(file)
write.table(x, file)

# RDS files
readRDS(file)
saveRDS(x, file)
```

dcast()
melt()

spread()
gather()

dplyr vs ggplot2 vs ggvis

Why?

- Because once you've understood one member of a family you understand them all
- You don't need to memorise special cases
- Easier to teach general principles which can be applied in multiple places

How?

- Punctuation
  (snake_case or camelCase: pick one!)
- Function names
  - Verb vs. noun
  - Tense, plural vs. singular
  - UK english
  - Think about autocomplete
- Argument names & order
- Object types

```
# It's not possible to consistent in every direction

# Would be nice if first argument was file
read.csv(file)
write.csv(x, file)

# Would be nice if first argument was a data frame
mutate(x)
filter(x)
write.csv(x, file)

# You can't reconcile conflicting notions
# of consistency. But important to be aware and
# consciously make tradeoff.
```

Pipeable

Goal: combine simple pieces with a standard tool

Why?

- Adds predictability across packages and across authors
- Learn once and apply in many situations
- Package doesn’t need to know about %>% to make use of it

How?

- The object being transformed should passed in as the first argument and returned from each object
- Must identify what the key object is.
- Can transform from one type of object to another, but must be clearly signposted

Examples

- **dplyr**: data frames
- **ggvis**: visualisations
- **gistr**: GitHub gists
- **lowliner**: vectors
- **tidyr**: messy data → tidy data
- **rvest**: html page, lists of nodes, single nodes
- **analogsea**: droplets, actions

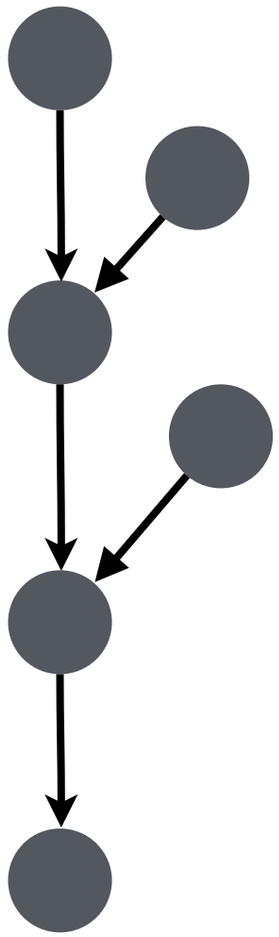
magrittr

Extensions and limitations

YES

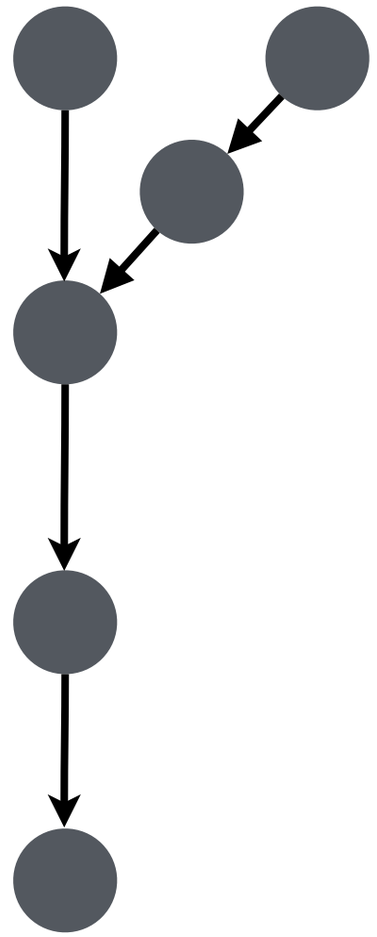
Linear sequence is
easiest to understand

YES



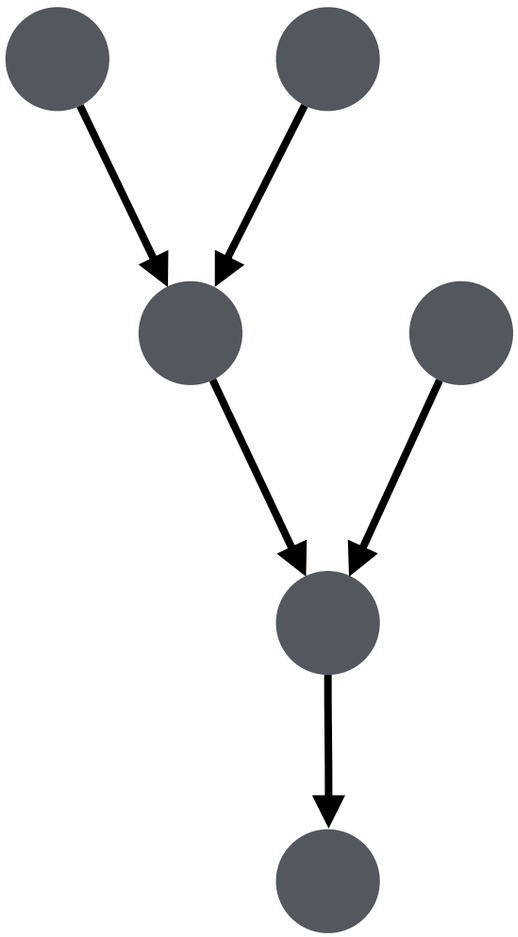
Sometimes other
objects are merged in

YES



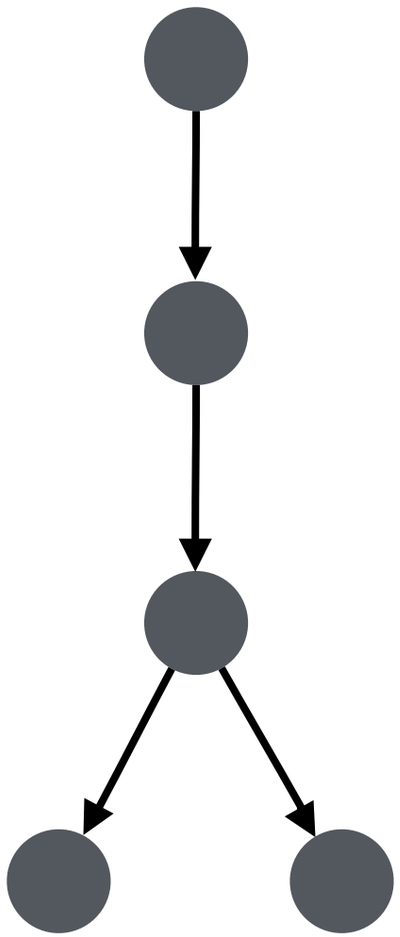
Can bring in sub-
pipes, in moderation

NO



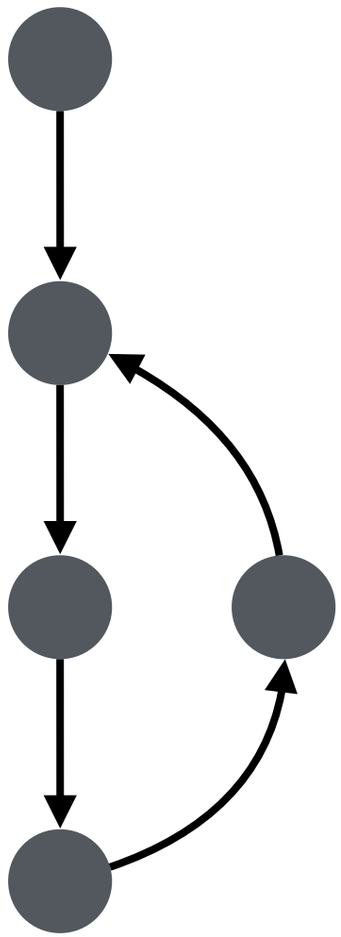
There is one primary path

NO

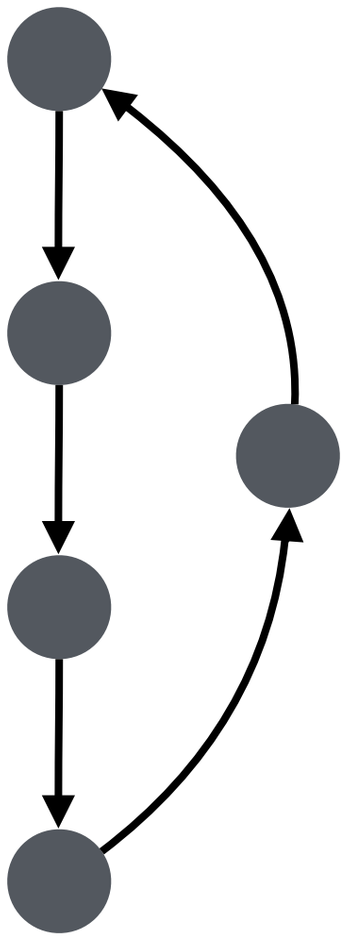


One output

NO

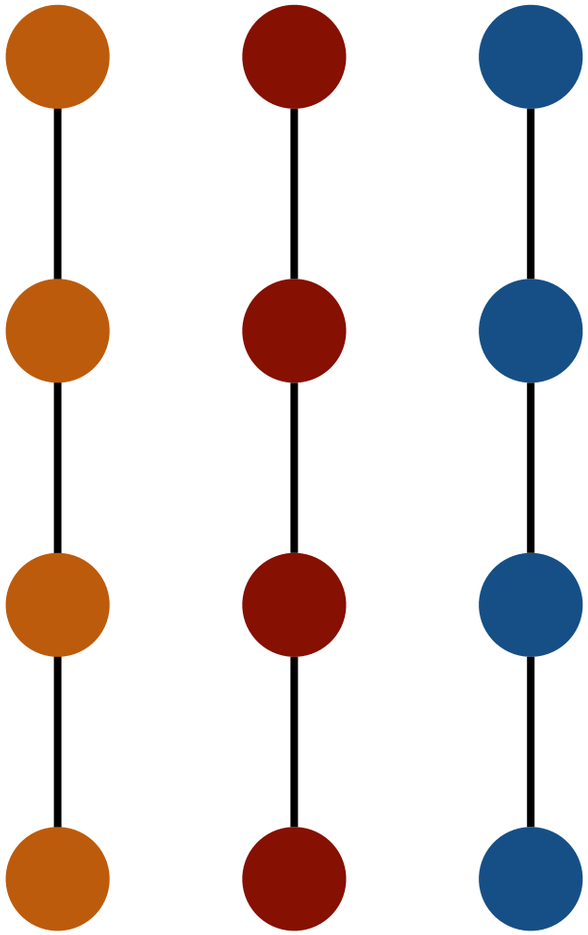


No cycles

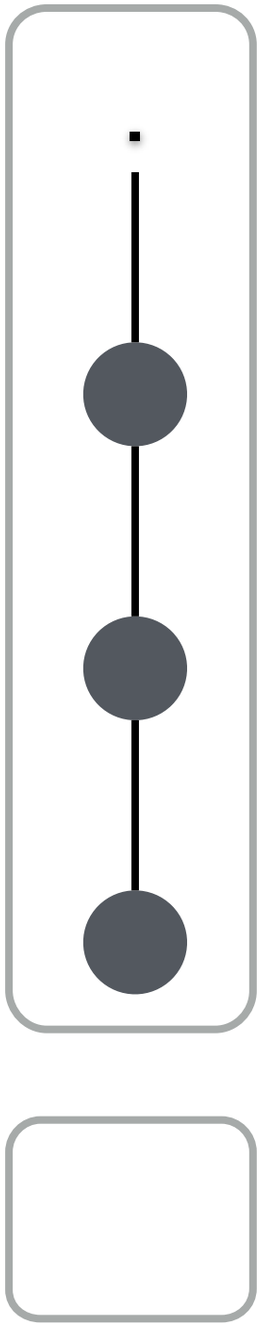


One exception: replace the initial element with %<>%

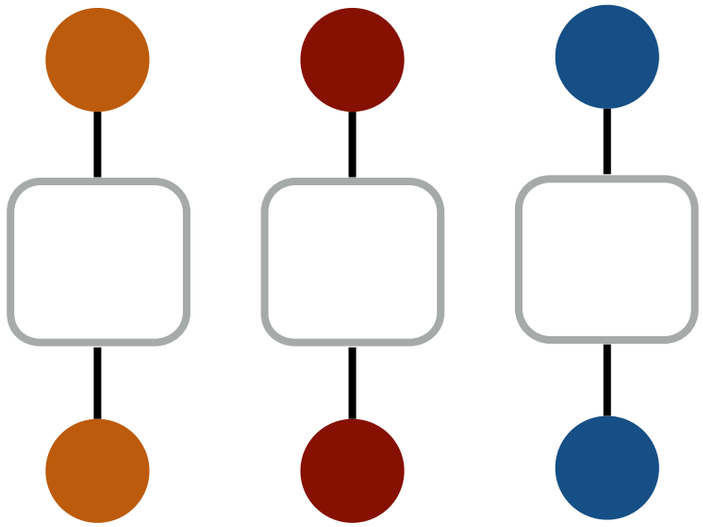
```
mtcars %<>% mutate(
  displ = displ * 0.0164
)

# Shorthand for
mtcars <-  mtcars %>% mutate(
  displ = displ * 0.0164
)
```



Frequently repeating the same pipeline with different input data?



Make a function!



```
to_metric <- . %>% mutate(
  displ = displ * 0.0164
)
mtcars %>% to_metric()
mtcars %<>% to_metric()

# Shorthand for
to_metric <- function(x) {
  x %>% mutate(
    displ = displ * 0.0164
  )
}
```

Conclusion

To make your own fluent interfaces

- Make simple functions that are easily understood in isolation
- Make sure they all work the same way. Think about verbs & nouns.
- Combine them together with %>%

Questions?

More about magrittr

https://github.com/smbache/magrittr