Pure, predictable, pipeable: creating fluent interfaces with R

Hadley Wickham
@hadleywickham
Chief Scientist, RStudio

November 2015

magrittr::

%>%

```
foo_foo <- little_bunny()

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

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

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

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

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

```
# 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
# Make your pure functions purr with purrr
# (hadley + lionel-)
library(purrr)

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

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

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

```
# Ensure objects are of correct type (smbache)
library(ensurer)
output <-
  some_computation() %>%
  ensure_that(is.numeric(.), NCOL(.) == NROW(.)) %>%
  some_more_computation() %>%
ensure_that(is.data.frame(.)) %>%
still_more_computation() %>%
ensure_that(all(.$x) > 0)
```

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()
S4
```

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

Type-stability

- Similar idea is type-stability: a function should always return the same type of thing.
- Good idea most of the time. But if all functions were type stable, `$` couldn't work!
- Great for interactive exploration, makes writing functions harder.

```
df <- data.frame(
  a = 1L,
  b = 1.5,
  y = Sys.time(),
  z = ordered(1)
)
```

```
sapply(df[1:4], class)
sapply(df[1:2], class)
sapply(df[3:4], class)
sapply(df[0], class)
```

Predictable

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

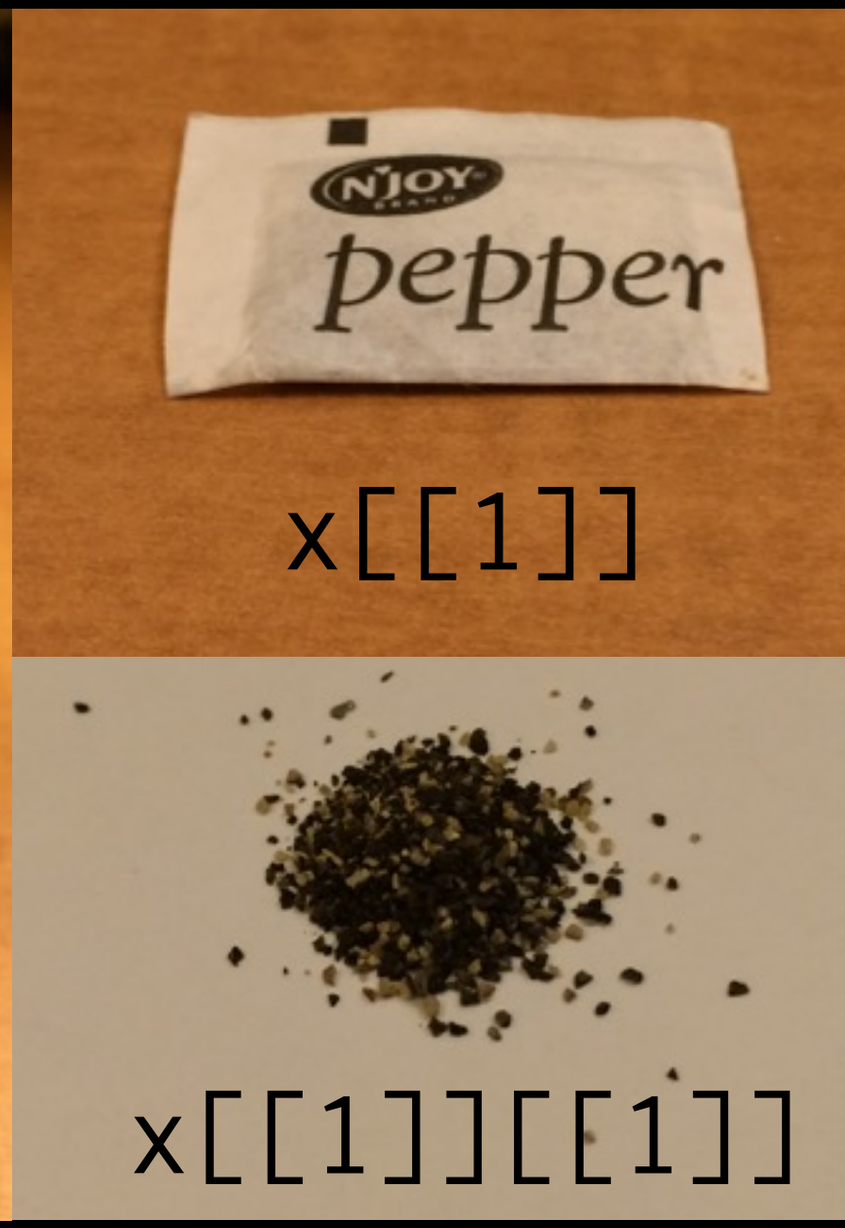
```
#rstats #wat


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, x, ...)
strsplit(x, split, ...)
substr(x, start, stop)
```

```
# I/O
read.csv(file) / write.csv(x, file)
read.table(file) / write.table(x, file)
readRDS(file) / saveRDS(x, file)

# Subsetting
x[1] vs x[[1]]
x[1, , drop = FALSE] vs x[1, ]
```

x

x[1]



x[[1]]

x[[1]][[1]]

Why?

- Learn once; apply many times
- Don’t need to memorise special cases
- Easier to teach
- Can fit all the main ideas on one piece of paper = cheatsheet

How?

- Punctuation
  (snake_case or camelCase: pick one!)
- Function names
  - Verb vs. noun
  - Plural vs. singular
  - UK vs. US 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 axes
# of consistency. Important to be aware and
# consciously make tradeoff.
# (dplyr vs ggvis)
```

Pipeable

Goal: combine simple pieces with a standard tool

(NB: this is a terrible name because it really has nothing to do with piping**)**

Why?

- Be predictable across packages/ authors; not just within.
- Learn once and apply in many situations.
- Peripherally related to piping; %>% most effective when everything works the same way.

How?

- Data should be the first argument.
- Use NSE judiciously; better to use (one sided) formulas instead.
- Only provide methods for `[[`, `[`, `+`, etc if v. good fit

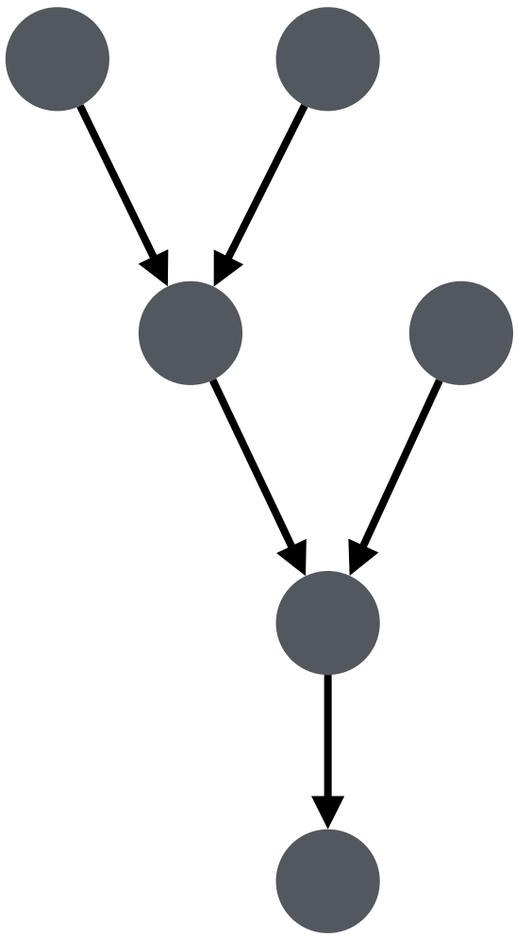
```
library(ggplot2)

qplot(mpg, wt, data = mtcars)
qplot(mtcars, ~mpg, ~wt)

ggplot(mtcars,  aes(mpg, wt)) +
  geom_point()
geom_point() + ggplot(mtcars,  aes(mpg, wt))
ggplot(mtcars, aes(~mpg, ~wt)) %>%
  geom_point()
geom_point(ggplot(mtcars, aes(mpg, wt)))

# One reason I don't like existing API
ggsave(ggplot(mtcars, aes(mpg, wt)) +
  geom_point())
```

```r
# base R
# https://developer.r-project.org/nonstandard-eval.pdf

lm(mpg ~ wt, data = mtcars, weight = n)
lm(mtcars, resp = ~mpg, pred = ~wt, weight = ~n)

subset(mpg, cyl == 4)
subset(mpg, ~cyl == 4)

# ???
cor(mpg$wt, mpg$cyl)
cor(mpg["wt", "cyl"])
cor(mpg, list(~wt, ~cyl))
```

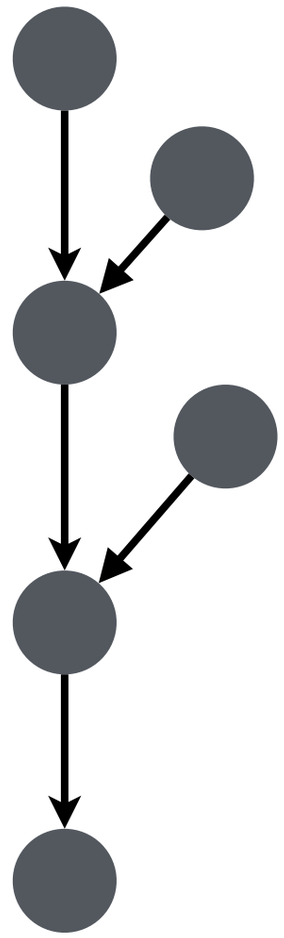
```
library(rvest)
lego_movie <- read_html("http://www.imdb.com/title/
tt1490017/")

# Had originally contemplated:
poster <- lego_movie[css("#img_primary")]["src"]
```

```
# Better to have special extractor functions
# than to use [[. What would it extract?
poster <- lego_movie %>%
  html_nodes("#img_primary img") %>%
  html_attr("src")
```

NO



YES



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