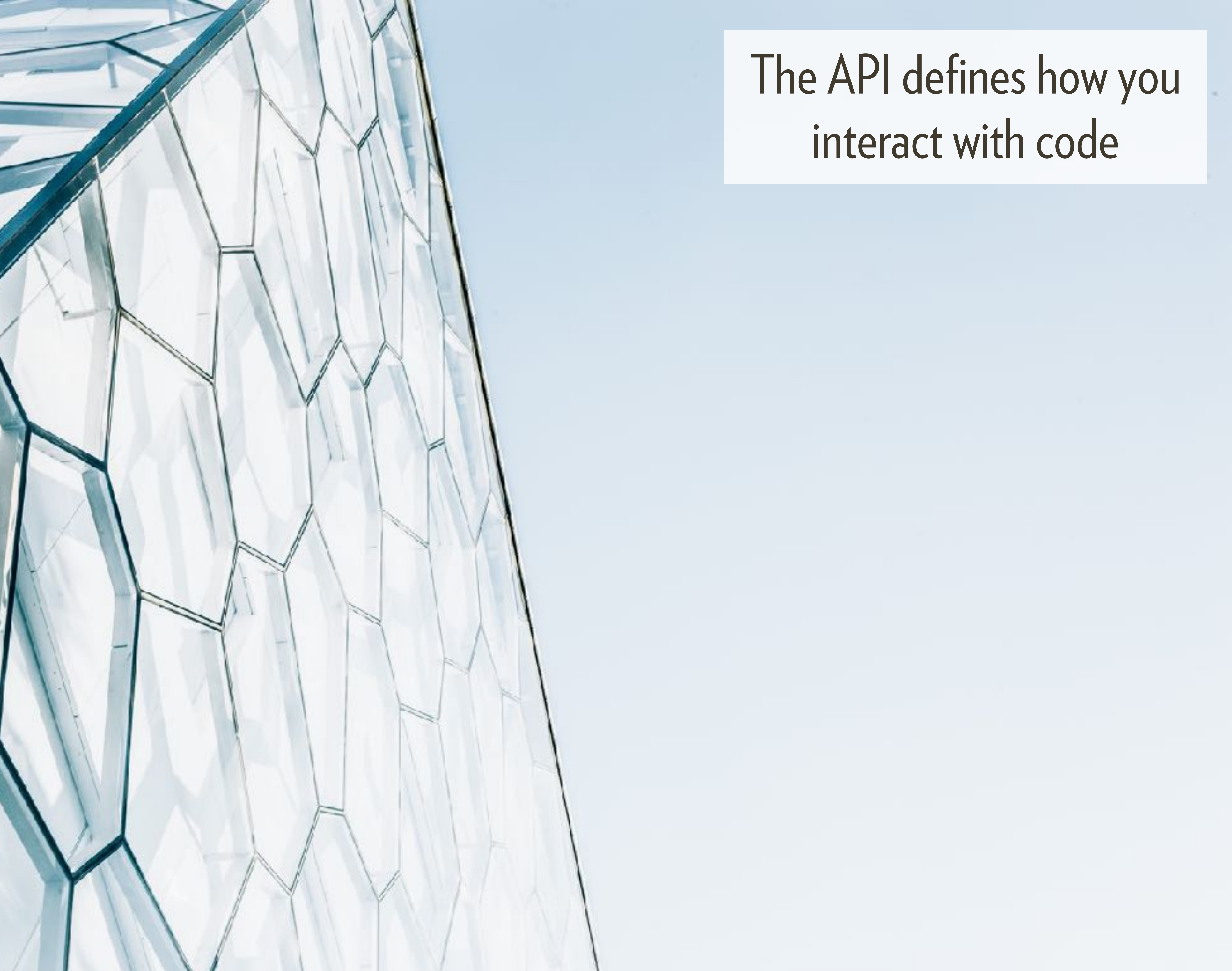
API design

January 2018

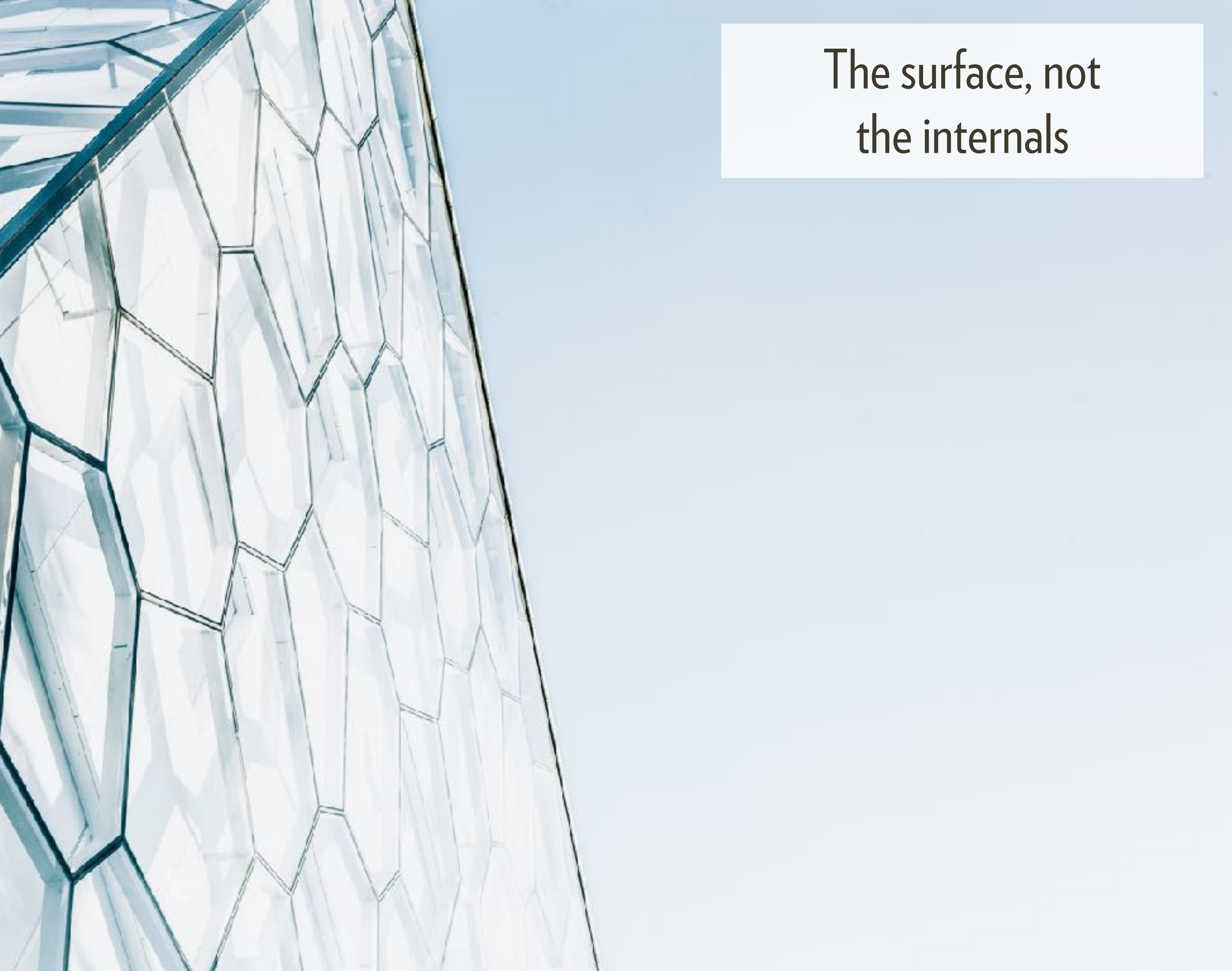
Hadley Wickham
@hadleywickham
Chief Scientist, **RStudio**

The API defines how you interact with code



https://unsplash.com/photos/QeJuMpjCHQk

The surface, not the internals



https://unsplash.com/photos/QeJuMpjCHQk

Case study

What makes base R functions hard to learn?

```
strsplit(x, split, ...)
grep(pattern, x, value = FALSE, ...)
grepl(pattern, x, ...)
sub(pattern, replacement, x, ...)
gsub(pattern, replacement, x, ...)
regexpr(pattern, text, ...)
gregexpr(pattern, text, ...)
regexec(pattern, text, ...)
subset(x, start, stop)
nchar(x, type, ...)
```

A few issues

Names: Function names have no common theme, and no common prefix. Names are concise at expense of expressiveness. Sometimes text, sometimes x.

Pipeability: Argument names & order are not consistent, and data isn't the first argument. Can't feed output of gregexpr() into substr()

Type stability: grep() is not type stable: can return string or integer. Can't feed output of gregexpr() into substr()

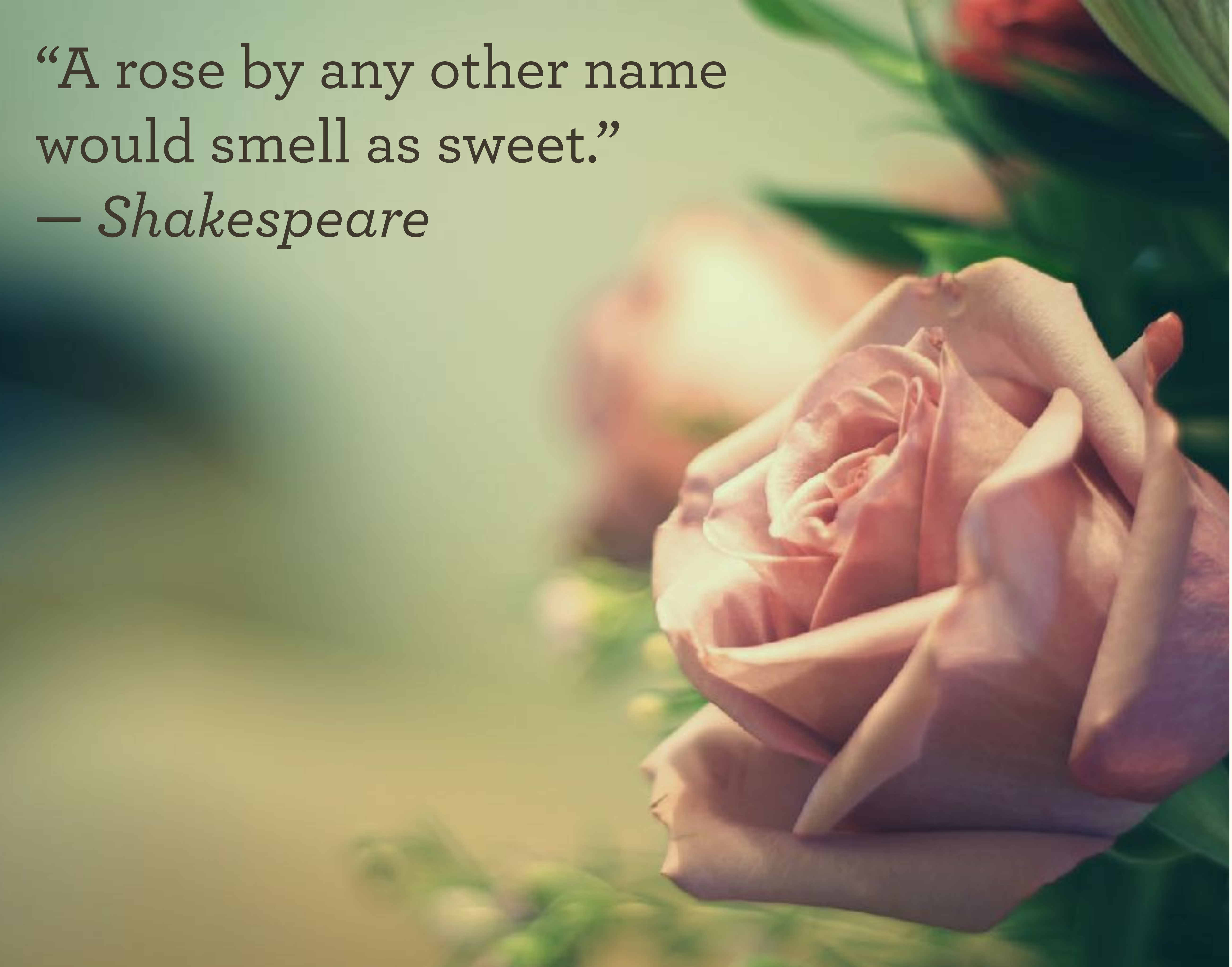
Each individual problem is small

https://unsplash.com/photos/HiiQsQcyO8g

“Each [function] is perfect the way it is .... and it can use a little improvement.”

—*Shunryu Suzuki*

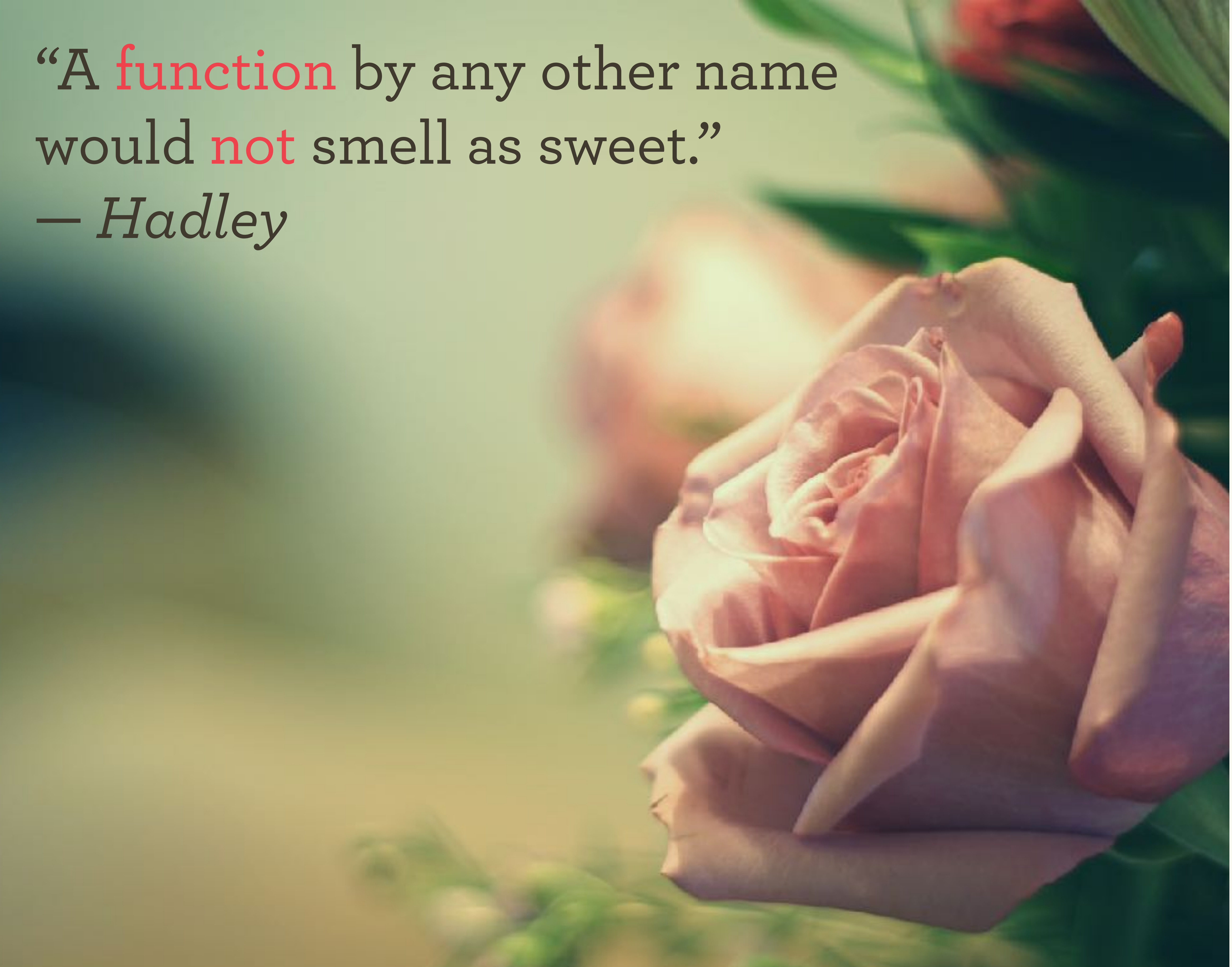
Carefully contemplate names



“A rose by any other name
would smell as sweet.”
— *Shakespeare*

https://unsplash.com/photos/wwokZzl_gcw

"A function by any other name would not smell as sweet."
— *Hadley*



https://unsplash.com/photos/wwokZzI_gcw

Principle:

Use prefixes to group related functions together

stringr uses consistent prefixes

```
str_split()
str_detect()
str_locate()
str_replace()

# Uses suffixes for variations on a theme
str_locate_all()
str_replace_all()
```

Principle:

Whenever you can give something an informative name, you should

stringr uses evocative verbs

```
str_split()
str_detect()
str_locate()
str_subset()
str_extract()
str_replace()

# But good verbs don't always exist
str_to_lower()
str_to_upper()
```

General advice

Be consistent!

Function names should be generally be verbs.

Prefer specific to general; concrete to abstract.

Avoid short names; err on the side of expressiveness.

Avoid names that differ in UK/US dialects.

Avoid names used in base R, or by similar packages.

You might get it wrong the first time

Your turn

Brainstorm functions that violate these principles (particularly within the tidyverse!)

Plan for pipes

Why is the pipe useful?

```r
library(dplyr)
by_dest <- group_by(flights, dest)
dest_delay <- summarise(by_dest,
  delay = mean(dep_delay, na.rm = TRUE),
  n = n()
)
big_dest <- filter(dest_delay, n > 100)
arrange(big_dest, desc(delay))
```

But naming is hard work

```
foo <- group_by(flights, dest)
foo <- summarise(foo,
  delay = mean(dep_delay, na.rm = TRUE),
  n = n()
)
foo <- filter(foo, n > 100)
arrange(foo, desc(delay))
```

But naming is hard work

```
foo1 <- group_by(flights, dest)
foo2 <- summarise(foo1,
  delay = mean(dep_delay, na.rm = TRUE),
  n = n()
)
foo3 <- filter(foo2, n > 100)
arrange(foo2, desc(delay))
```

Alternatively, you *could* nest function calls

```
arrange(
  filter(
    summarise(
      group_by(flights, dest),
      delay = mean(dep_delay, na.rm = TRUE),
      n = n()
    ),
    n > 100
  ),
  desc(delay)
)
```

magrittr provides a third option

%>%

No intermediaries; read from left-to-right

```
flights %>%
  group_by(dest) %>%
  summarise(
    delay = mean(dep_delay, na.rm = TRUE),
    n = n()
  ) %>%
  filter(n > 100) %>%
  arrange(desc(delay))
```

	Read left-to-right	Can omit intermediate names	Non-linear

`y <- f(x)`
`g(y)`
	✅		✅

`g(f(x))`		✅	✅

`x %>%`
`  f() %>%`
`  g()`
	✅	✅	

Principle:

Data arguments should come first

Most arguments fall in one of two classes

Data	Details
Required	Optional
Core data	Additional options
Often vectorised	
Often called x or data	

Your turn

Which are the data arguments in grepl()?
Which are the details?

Which are the data arguments in strsplit()?
Which are the details?

Which are the data arguments in substr()?
Which are the details?

And consistent argument order + names

```
str_split(string, pattern)
str_detect(string, pattern)
str_locate(string, pattern)
str_replace(string, pattern, replacement)
```

Principle:
Match outputs and inputs

Your turn

How does regexpr() return location of match?
How does gregexpr() return location?

How does substr() take locations?
What's difference between substr() and substring()

Output of regexp() not compatible with substr()

```
x <- c("bbaab", "bbb", "bbaaba")
regexpr("a+", x)

loc <- regexpr("a", x)
substr(x, loc, loc + attr(loc, "match.length"))

# And only works because this returns ""
substr(x, -1, -2)
```

Equivalent stringr code is much simpler

```r
library(stringr)

x <- c("bbaab", "bbb", "bbaaba")
str_sub(x, str_locate(x, "a+"))
```