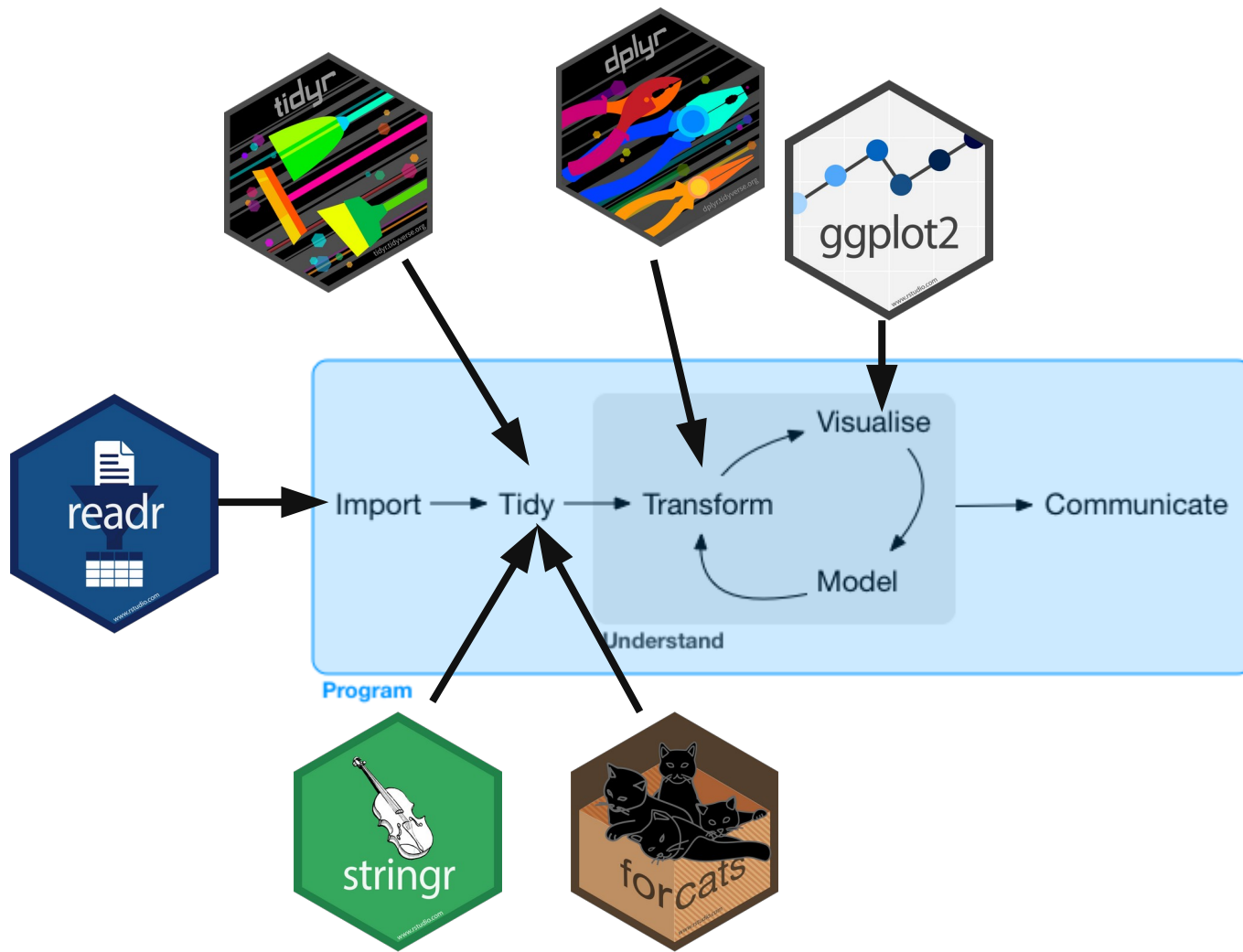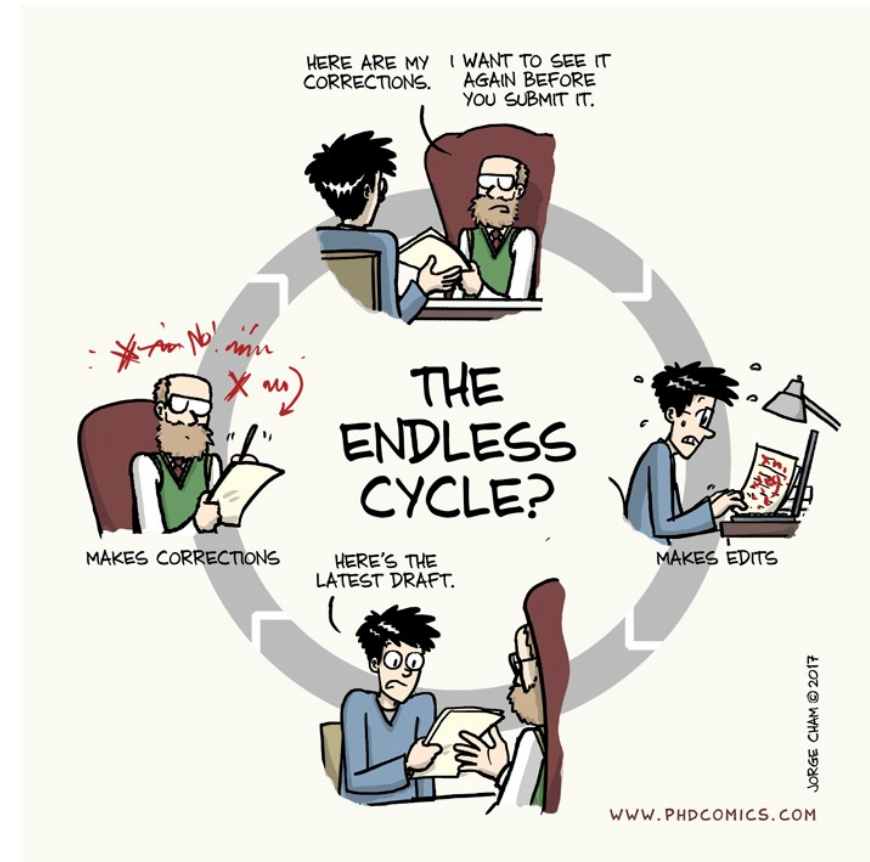# GG606

Pipes & Functions

Program

- Programming produces code
- Code is communication

Program

- Programming produces code
- Code is communication
  - Other people
  - Past & future self
- →getting better at programming means getting better at communicating
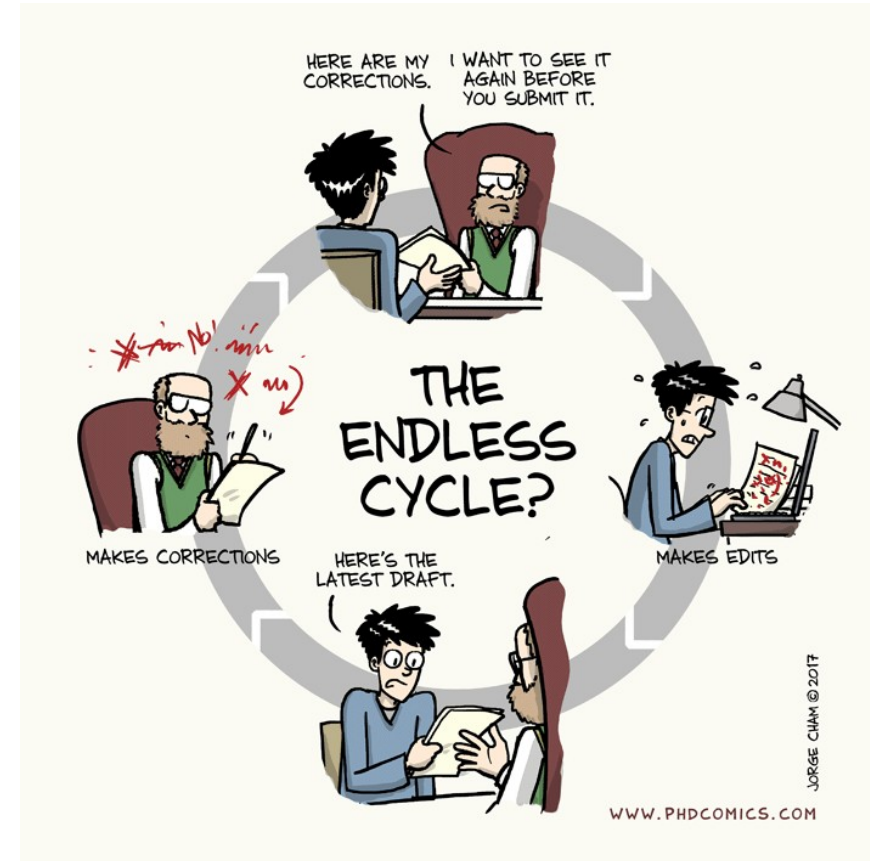
Prose

- How many (rounds of) edits for your thesis?

Prose

- How many (rounds of) edits for your thesis?

- Obvious what your code does?

- Rewrite while ideas are fresh

Time

- Balance of time/effort now vs later

- Pipes: %>% and |>

- Copy-and-paste

- Functions

Time

- Applies equally to spreadsheets, R, Matlab, Python, Julia

- Plan ahead as much as you can

magrittr pipe

- `tidyverse` packages load `magrittr::%>%`
- Great example using pseudocode
  - *Note: ask about pseudocode*

> Little bunny Foo Foo
> Went hopping through the forest
> Scooping up the field mice
> And bopping them on the head

This is a popular Children's poem that is accompanied by hand actions.

> Little bunny Foo Foo
> Went hopping through the forest
> Scooping up the field mice
> And bopping them on the head

This is a popular Children's poem that is accompanied by hand actions.

How would we do this in (pseudo)code?

Four ways in code

- Verbs are functions: hop( ), scoop( ), bop( )
  -
  -
  -
  -

Little bunny Foo Foo
Went hopping through the forest
Scooping up the field mice
And bopping them on the head

Four ways in code

- Verbs are functions: hop( ), scoop( ), bop( )
  1) Save each intermediate step as a new object
  2) Overwrite the original many times
  3) Compose functions
  4) Use the pipe

Little bunny Foo Foo
Went hopping through the forest
Scooping up the field mice
And bopping them on the head

1) Intermediate steps

- `foo_foo_1 ← hop(foo_foo, through = forest)`
- `foo_foo_2 ← scoop(foo_foo_1, up = field_mice)`
- `foo_foo_3 ← bop(foo_foo_2, on = head)`

1) Intermediate steps

- `foo_foo_1 ← hop(foo_foo, through = forest)`
- `foo_foo_2 ← scoop(foo_foo_1, up = field_mice)`
- `foo_foo_3 ← bop(foo_foo_2, on = head)`
- Natural names for intermediates?
- Cluttered, easy to make mistakes

2) Overwrite original

- `foo_foo ← hop(foo_foo, through = forest)`
- `foo_foo ← scoop(foo_foo, up = field_mice)`
- `foo_foo ← bop(foo_foo, on = head)`

2) Overwrite original

- `foo_foo ← hop(foo_foo, through = forest)`
- `foo_foo ← scoop(foo_foo, up = field_mice)`
- `foo_foo ← bop(foo_foo, on = head)`
- How to debug?
- Repetition obscures what's being changed

3) Function composition

- ```
  bop(
    scoop(
      hop(foo_foo, through = forest),
      up = field_mice
    ),
    on = head
  )
  ```

3) Function composition

- ```
  bop(
    scoop(
      hop(foo_foo, through = forest),
      up = field_mice
    ),
    on = head
  )
  ```

- Reading inside-out, right-to-left

- Arguments far apart
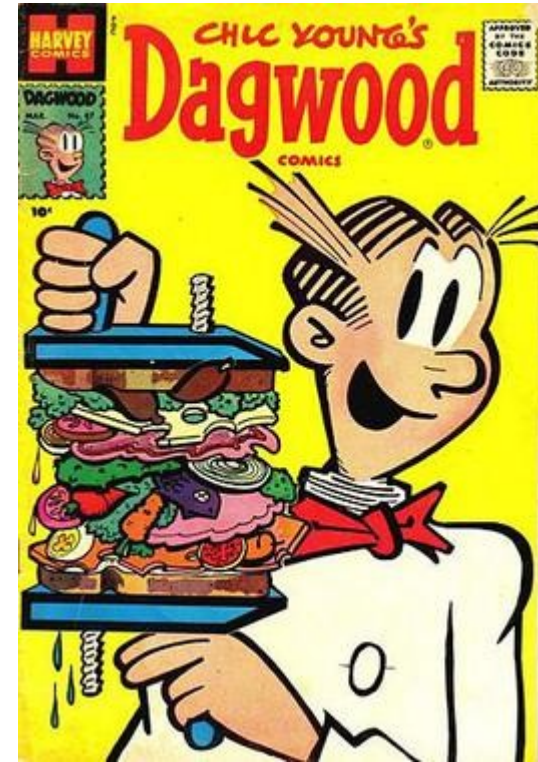
4) Pipe

- ```
  foo_foo %>%
    hop(through = forest) %>%
    scoop(up = field_mice) %>%
    bop(on = head)
  ```

Python: sspipe, toolz

4) Pipe

- ```
  foo_foo %>%
      hop(through = forest) %>%
      scoop(up = field_mice) %>%
      bop(on = head)
  ```

- Focus on verbs, not nouns

- Easily read

4) Pipe

- ```
  foo_foo %>%
    hop(through = forest) %>%
    scoop(up = field_mice) %>%
    bop(on = head)
  ```
  Call this

- ```
  my_pipe ← function(.) {
    . ← hop(., through = forest)
    . ← scoop(., up = field_mice)
    bop(., on = head)
  }
  ```
  Get this

- ```
  my_pipe(foo_foo)
  ```
  Equivalent to this
  Pays no attention to current env

Other `magrittr` tools

```
rnorm(100) %>%
  matrix(ncol = 2) %>%
  plot() %>%
  str()
```

```
rnorm(100) %>%
  matrix(ncol = 2) %T>%
  plot() %>%
  str()
```

Tee pipe returns left-hand side instead of right-hand side

Other `magrittr` tools

```
mtcars %$%
  cor(disp, mpg)
```

```
> mtcars %>%
+     cor(disp, mpg)
Error in cor(., disp, mpg) : invalid 'use' argument
In addition: Warning message:
In if (is.na(na.method)) stop("invalid 'use' argument") :
  the condition has length > 1 and only the first element will be used
```

Exposition pipe 'explodes' out variables for functions that need vectors

Other `magrittr` tools

```
mtcars ← mtcars %>%
  transform(cyl = cyl * 2)

mtcars %<>% transform(cyl = cyl * 2)
```

Assignment pipe combines ← with %>%

functions

Why functions

- Good name makes code easier to understand

- Only update on one place (not many)

- Eliminate changes of making incidental mistakes when copy-pasting

- True across all languages

When should you write a function?

- ```
  df ← tibble::tibble(
    a = rnorm(10),
    b = rnorm(10),
    c = rnorm(10),
    d = rnorm(10)
  )
  ```
- ```
  df$a ← (df$a - min(df$a, na.rm = TRUE)) /
    (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))
  ```
- ```
  df$b ← (df$b - min(df$b, na.rm = TRUE)) /
    (max(df$b, na.rm = TRUE) - min(df$a, na.rm = TRUE))
  ```
- ```
  df$c ← (df$c - min(df$c, na.rm = TRUE)) /
    (max(df$c, na.rm = TRUE) - min(df$c, na.rm = TRUE))
  ```
- ```
  df$d ← (df$d - min(df$d, na.rm = TRUE)) /
    (max(df$d, na.rm = TRUE) - min(df$d, na.rm = TRUE))
  ```

Good example
What does it do?

When should you write a function?

- `df$a ← (df$a - min(df$a, na.rm = TRUE)) / (max(df$a, na.rm = TRUE) - min(df$a, na.rm = TRUE))`

How many inputs

When should you write a function?

- ```
  (df$a - min(df$a, na.rm = TRUE)) /
    (max(df$a, na.rm = TRUE) -
    min(df$a, na.rm = TRUE))
  ```

- ```
  x ← df$a
  ```

How many inputs? 1
Duplication?

When should you write a function?

- ```
  (df$a - min(df$a, na.rm = TRUE)) /
    (max(df$a, na.rm = TRUE) -
    min(df$a, na.rm = TRUE))
  ```

- ```
  x ← df$a
  ```

- ```
  (x - min(x, na.rm = TRUE)) /
    (max(x, na.rm = TRUE) -
    min(x, na.rm = TRUE))
  ```

How many inputs? 1
Duplication?

When should you write a function?

- `rng ← range(x, na.rm = TRUE)`
- `(x - rng[1]) / (rng[2] - rng[1])`

Intermediate calculations into name variable
Still work?

When should you write a function?

- ```
rescale01 ← function(x) {
    rng ← range(x, na.rm = TRUE)
    (x - rng[1]) / (rng[2] - rng[1])
}
```

- `rescale01(c(0, 5, 10))`

When should you write a function?

- Pick a unique and descriptive name: `rescale01`

- Inputs to the function, arguments, inside `function()`

- Your code in the body of the function with `{}` immediately after `function( ... )`

Make function after making it work with simple input

When should you write a function?

- `rescale01(c(0, 5, 10))`

- `rescale01(c(-10, 0, 10))`

- `rescale01(c(1, 2, 3, NA, 5))`

Make function after making it work with simple input
Informal tests, Unit testing

When should you write a function?

- df$a ← rescale01(df$a)

- df$b ← rescale01(df$b)

- df$c ← rescale01(df$c)

- df$d ← rescale01(df$d)

One set of code
Easy to read

When should you write a function?

- `rescale01(c(1:10, Inf))`

```
> rescale01(c(1:10, Inf))
 [1]   0   0   0   0   0   0   0   0   0 NaN
```

Easy to fix

When should you write a function?

- `rescale01(c(1:10, Inf))`

- ```
  rescale01 ← function(x) {
    rng ← range(x, na.rm = TRUE, finite = TRUE)
    (x - rng[1]) / (rng[2] - rng[1])
  }
  ```

- `rescale01(c(1:10, Inf))`

Easy to fix

Make Functions

- `mean(is.na(x))`
- `x / sum(x, na.rm = TRUE)`
- `sd(x, na.rm = TRUE) / mean(x, na.rm = TRUE)`

Homework

- Functions for variance and skewness

$$\mathrm{Var}(x) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

$$\mathrm{Skew}(x) = \frac{\frac{1}{n-2}\left(\sum_{i=1}^{n}(x_i - \bar{x})^3\right)}{\mathrm{Var}(x)^{3/2}}$$