ONDREJ IVANIČ

WRITING BETTER R CODE

"C" LIKE LANGUAGE TRAP

"C" LIKE LANGUAGE TRAP

- Indexing is confusing: [, [[, and $
  - Functions with surprising behaviours
  - Lowest index is one not zero
- No scalar type - everything is vector / list

"C" LIKE LANGUAGE TRAP

- Indexing is confusing: [, [[, and $
  - Functions with surprising behaviours
  - Lowest index is one not zero
    - No scalar type - everything is vector / list
- Troublesome (for)loops
  - Unnecessary looping over data
  - Never use length() / nrow() in loops: use seq_along() or seq_len()

"C" LIKE LANGUAGE TRAP

- Indexing is confusing: [, [[, and $
  - Functions with surprising behaviours
  - Lowest index is one not zero
    - No scalar type - everything is vector / list
- Troublesome (for)loops
  - Unnecessary looping over data
  - Never use length() / nrow() in loops: use seq_along() or seq_len()
- stringsAsFactors default is TRUE

(TOO) FLEXIBLE SUBSETTING

- x <- 1:10
  x[1]
  [1] 1

(TOO) FLEXIBLE SUBSETTING

- x <- 1:10
  x[1]
  [1] 1
- x[2.0]
  [1] 1

(TOO) FLEXIBLE SUBSETTING

- x <- 1:10
  x[1]
  [1] 1

- x[2.0]
  [1] 1

- x[c(1,2,6)]
  [1] 1 2 6

(TOO) FLEXIBLE SUBSETTING

- x <- 1:10
  x[1]
  [1] 1

- x[2.0]
  [1] 1

- x[c(1,2,6)]
  [1] 1 2 6

- x[-3]
  [1] 1 2 4 5 6 7 8 9 10

(TOO) FLEXIBLE SUBSETTING - WRITING BETTER R CODE

- x <- 1:10

(TOO) FLEXIBLE SUBSETTING - WRITING BETTER R CODE

- x <- 1:10
- x[c(TRUE, FALSE, FALSE)]

```
[1]  1  4  7 10
```

(TOO) FLEXIBLE SUBSETTING - WRITING BETTER R CODE

- x <- 1:10
- x[c(TRUE, FALSE, FALSE)]
  [1] 1 4 7 10
- x[c(TRUE, FALSE, TRUE)]
  [1] 1 3 4 6 7 9 10

(TOO) FLEXIBLE SUBSETTING - WRITING BETTER R CODE

- x <- 1:10
- x[c(TRUE, FALSE, FALSE)]
  [1] 1 4 7 10
- x[c(TRUE, FALSE, TRUE)]
  [1] 1 3 4 6 7 9 10
- x[NA]
  [1] NA NA NA NA NA NA NA NA NA NA

(TOO) FLEXIBLE SUBSETTING - WRITING BETTER R CODE

- x <- 1:10
- x[c(TRUE, FALSE, FALSE)]
  [1] 1 4 7 10
- x[c(TRUE, FALSE, TRUE)]
  [1] 1 3 4 6 7 9 10
- x[NA]
  [1] NA NA NA NA NA NA NA NA NA NA
- x[NA_integer_]
  [1] NA

(TOO) FLEXIBLE SUBSETTING - WRITING BETTER R CODE

- x <- 1:10
- x[c(TRUE, FALSE, FALSE)]
  [1] 1 4 7 10
- x[c(TRUE, FALSE, TRUE)]
  [1] 1 3 4 6 7 9 10
- x[NA]
  [1] NA NA NA NA NA NA NA NA NA NA
- x[NA_integer_]
  [1] NA
- x[c(NA, NA, 1)]
  [1] NA NA 1

(TOO) FLEXIBLE SUBSETTING - WRITING BETTER R CODE

- x <- 1:10
- x[c(TRUE, FALSE, FALSE)]
  [1] 1 4 7 10
- x[c(TRUE, FALSE, TRUE)]
  [1] 1 3 4 6 7 9 10
- x[NA]
  [1] NA NA NA NA NA NA NA NA NA NA
- x[NA_integer_]
  [1] NA
- x[c(NA, NA, 1)]
  [1] NA NA 1
- x[NULL]
  integer(0)

- `[` and `[[` are functions!
- Functions can take arguments:

  x[i]

  x[i, j, ... , drop = TRUE]

  x[[i, exact = TRUE]]

  x[[i, j, ..., exact = TRUE]]

  x$name
- type ?`[` or ?Extract for help about those functions

TROUBLESOME (FOR)LOOPS

- Non-linear code reading path
- Result should be pre-allocated
  - $O(n^2)$ vs $O(n)$ performance
- Inside the loop code modifies outside the loop data
- Vectorise when possible
  ifelse(), sum(log(x)), ...
- Loop hiding using *apply()

- Zero (empty) length vector problem

```
x <- NULL
for(i in 1:length(x)) {
  print(x[i])
}
NULL
```

- Zero (empty) length vector problem

```
x <- NULL
for(i in 1:length(x)) {
  print(x[i])
}
```

```
NULL
```

- seq_along() generates proper sequence

```
for(i in seq_along(x)) {
  print(x[i])
}
```

```
(no output)
```

- data.frame is a list with “data.frame” class

```
x <- data.frame(x = 1:10, y = 10:1)
str(unclass(x))
List of 2
 $ x: int [1:10] 1 2 3 4 5 6 7 8 9 10
 $ y: int [1:10] 10 9 8 7 6 5 4 3 2 1
 - attr(*, "row.names")= int [1:10] 1 2 3 4 5 6 7 8 9 10
```

- data.frame is a list with “data.frame” class

```
x <- data.frame(x = 1:10, y = 10:1)
str(unclass(x))
List of 2
 $ x: int [1:10] 1 2 3 4 5 6 7 8 9 10
 $ y: int [1:10] 10 9 8 7 6 5 4 3 2 1
 - attr(*, "row.names")= int [1:10] 1 2 3 4 5 6 7 8 9 10
```

- length() returns number of columns

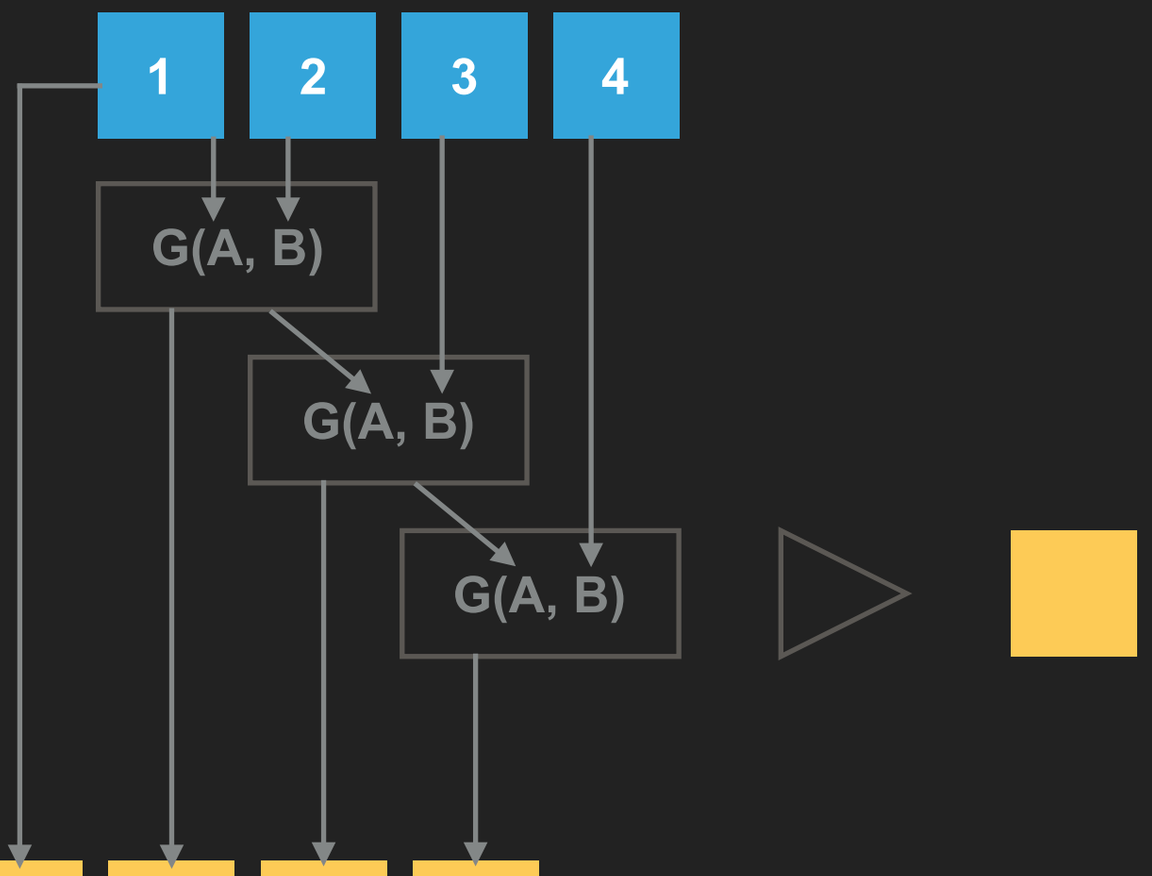
```
for(i in seq_len(nrow(x))) {
  print(x[i, ])
}
   x  y
1 1 10
...
```

- Data frames: use “dplyr”, “tidyr” and list columns
- Result is the same shape
  lapply, Reduce, map() - like functions, accumulate, cumsum, vectorised functions, …
- Result is one element
  Reduce, colMeans, sum, reduce, …
- Result is a subset of input
  Filter, keep, discard, …
- Result is anything else
  recursion or for-loop

Base R
purr: Functional Programming Tools

map(x, F)

reduce(x, G)



accumulate(x, G)

FUNCTIONAL PROGRAMMING

- Programming with functions
- Pure functions
  - result is stable and depends on arguments only
  - no side effects
- Impure functions are useful
  - Reading and writing files (I/O), graphics, and random numbers
- Isolate side effects into specialised functions and places
- Write function when there is repetition in the code

FUNCTION FACTORY

- Compose multiple functions

```
not_null <- function(x) !is.null(x)
not_null <- compose(`!`, is_null)
```

```
n_distinct <- compose(length, unique)
compose(length, unique)(NA, NA, 10, 10)
```

- Filing some arguments

```
row_apply <- function(x, ...) apply(x, MARGIN = 1, ...)
row_apply <- partial(apply, MARGIN = 1)
row_apply(matrix(1:9, nrow = 3), sum)

add_five <- partial(`+`, e2 = 5)
```

- Filing some arguments

```
row_apply <- function(x, ...) apply(x, MARGIN = 1, ...)
row_apply <- partial(apply, MARGIN = 1)
row_apply(matrix(1:9, nrow = 3), sum)

add_five <- partial(`+`, e2 = 5)
```

- Change argument domain

```
plus1 <- `+`
plus2 <- lift_dl(`+`)

identical(plus1(1, 1), plus2(list(1, 1)))
[1] TRUE
```

%>% (PIPE)

- moves data from one function to another

%>% (PIPE)

- moves data from one function to another
- left side (previous result) as the first argument
  x %>% f(y) ⇔ f(x, y)

%>% (PIPE)

- moves data from one function to another
- left side (previous result) as the first argument
  x %>% f(y) ⇔ f(x, y)
- Dot (.) is placeholder
  x %>% { left_join(other_data, .) }
  x %>% (function(df) left_join(other_data, df))
  left_join(other_data, x)

- Removes ugly function call nesting

```
plot(diff(log(rnorm(100, mean = 10))), col = "red")

rnorm(100, mean = 10) %>%
  log %>%
  diff %>%
  plot(col=“red")
```

- Removes ugly function call nesting

```
plot(diff(log(rnorm(100, mean = 10))), col = "red")

rnorm(100, mean = 10) %>%
  log %>%
  diff %>%
  plot(col=“red")
```

- Pipes are linear - complex (and long) flow is confusing

- Removes ugly function call nesting

```
plot(diff(log(rnorm(100, mean = 10))), col = "red")

rnorm(100, mean = 10) %>%
  log %>%
  diff %>%
  plot(col=“red")
```

- Pipes are linear - complex (and long) flow is confusing
- Unary functions from pipes

```
log_diff <- . %>% log %>% diff
```

EXAMPLE - WRITING BETTER R CODE

```r
library(tidyverse)
library(rpart)

data <- mtcars

tree_model <- function(formula, data) {
  function(depth, minsplit) {
    ctrl <- rpart.control(
      maxdepth = depth, minsplit = minsplit, cp = 1e-6
    )

    fit <- rpart(formula, data, method = "anova", control = ctrl)
    cp <- fit$cptable[which.min(fit$cptable[, “xerror"]), "CP"]

    prune(fit, cp)
  }
}

fit_model <- tree_model(mpg ~ cyl + disp + wt, data)

cv_error <- . %>% {tail(.[["cptable"]], 1)} %>% as.data.frame
```

EXAMPLE - WRITING BETTER R CODE

```
models <- cross_d(list(
  depth = 1:10, minsplit = c(10, 20, 50)
)) %>%
  mutate(
    model = pmap(list(depth, minsplit), fit_model)
    , cv_error = map(model, cv_error)
  ) %>%
  unnest(cv_error, .drop = FALSE) %>%
  filter(xerror > 0) %>%
  arrange(xerror)
```

EXAMPLE - WRITING BETTER R CODE

```
> models
# A tibble: 20 × 8
   depth minsplit       model           CP nsplit `rel error`    xerror       xstd
   <int>    <dbl>      <list>        <dbl>  <dbl>        <dbl>     <dbl>      <dbl>
1      8       10 <S3: rpart> 0.007273533      4   0.09320330 0.2414180 0.05670010
2      4       10 <S3: rpart> 0.000001000      5   0.08592977 0.2914916 0.07400746
3      6       10 <S3: rpart> 0.000001000      5   0.08592977 0.3077009 0.08165256
4      3       10 <S3: rpart> 0.000001000      4   0.09320330 0.3160728 0.07402147
5      5       10 <S3: rpart> 0.023249716      3   0.11645302 0.3300807 0.07425913
6      7       10 <S3: rpart> 0.000001000      5   0.08592977 0.3583280 0.10169631
7      8       20 <S3: rpart> 0.000001000      2   0.26753994 0.3607337 0.07544325
8      2       10 <S3: rpart> 0.000001000      2   0.15263644 0.3611394 0.07850712
9      9       10 <S3: rpart> 0.000001000      5   0.08592977 0.3618312 0.11484704
10    10       20 <S3: rpart> 0.000001000      2   0.26753994 0.3657437 0.07125878
11     6       20 <S3: rpart> 0.089334834      1   0.35687477 0.3872654 0.07633373
12    10       10 <S3: rpart> 0.007273533      4   0.09320330 0.3890450 0.11418850
13     5       20 <S3: rpart> 0.000001000      2   0.26753994 0.3910748 0.07826596
14     9       20 <S3: rpart> 0.000001000      2   0.26753994 0.4011769 0.08006036
15     2       20 <S3: rpart> 0.000001000      2   0.26753994 0.4062335 0.08125204
16     7       20 <S3: rpart> 0.000001000      2   0.26753994 0.4105033 0.07369474
17     1       20 <S3: rpart> 0.000001000      1   0.35687477 0.4370306 0.08118937
18     3       20 <S3: rpart> 0.000001000      2   0.26753994 0.4683397 0.09609614
19     4       20 <S3: rpart> 0.000001000      2   0.26753994 0.5234494 0.08651928
20     1       10 <S3: rpart> 0.000001000      1   0.34733879 0.6589797 0.12423875
```

EXAMPLE - WRITING BETTER R CODE

```r
library(purrr)

walk1 <- function(n) {
  x <- sample(c(-1, 1), n, replace = TRUE)
  for(i in seq_len(nn)[-1]) { # Skip first index
    x[i] <- x[i - 1] + x[i]
  }
  x
}

steps <- function(n) sample(c(-1, 1), n, replace = TRUE)

walk2 <- function(n) cumsum(steps(n))
walk3 <- function(n) accumulate(steps(n), `+`)
walk4 <- function(n) Reduce(`+`, steps(n), accumulate = TRUE)
```

TO SUM UP

- Use pure functions and keep side-effects in one place
- Hide for-loops with map/filter/reduce
- %>% is useful
- purrr, tidyr, and dplyr are good packages to master