R Functions

Robust code

What do these calls do?

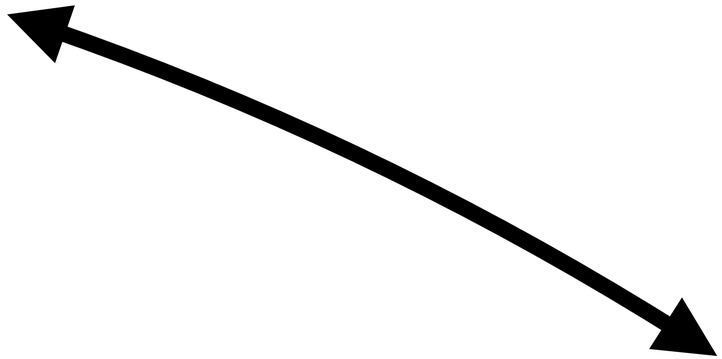
```
> df[, vars]

> subset(df, x == y)

> data.frame(x = "a")
```

Interactive analysis

Helpful



Programming

Strict

Three main problems

- Type-unstable functions
- Non-standard evaluation
- Hidden arguments

Throwing errors

```
> x <- 1:10

> stopifnot(is.character(x))
Error: is.character(x) is not TRUE
```

Throwing errors

```
> x <- 1:10

> stopifnot(is.character(x))
Error: is.character(x) is not TRUE
```

```
if (condition) {
  stop("Error", call. = FALSE)
}
```

Throwing errors

```
> x <- 1:10

> stopifnot(is.character(x))
Error: is.character(x) is not TRUE
```

```
if (condition) {
  stop("Error", call. = FALSE)
}
```

```
> if (!is.character(x)) {
    stop("`x` should be a character vector", call. = FALSE)
  }
Error: `x` should be a character vector
```

R Functions

Let’s practice!

R Functions

Unstable types

Surprises due to unstable types

- Type-inconsistent: the type of the return object depends on the input
- Surprises occur when you've used a type-inconsistent function inside your own function
- Sometimes lead to hard to decipher error messages

What will `df[1, ]` return?

```
> df <- data.frame(z = 1:3, y = 2:4)
> str(df[1, ])
'data.frame':1 obs. of  2 variables:
 $ z: int 1
 $ y: int 2

> df <- data.frame(z = 1:3)
> str(df[1, ])
 int 1
```

[is a common source of surprises

```
> last_row <- function(df) {
      df[nrow(df), ]
  }

> df <- data.frame(x = 1:3)

# Not a row, just a vector
> str(last_row(df))
 int 3
```

Two common solutions for [

```
> last_row <- function(df) {
    df[nrow(df), , drop = FALSE]
  }

> df <- data.frame(x = 1:3)

> str(last_row(df))
'data.frame':1 obs. of  1 variable:
 $ x: int 3
```

- Use `drop = FALSE`: `df[x, , drop = FALSE]`
- Subset the data frame like a list: `df[x]`

What to do?

- Write your own functions to be type-stable
- Learn the common type-inconsistent functions in R:
  `[`, `sapply`
- Avoid using type-inconsistent functions inside your own functions
- Build a vocabulary of type-consistent functions

R Functions

Let’s practice!

R Functions

Non-standard evaluation

What is non-standard evaluation?

```
> subset(mtcars, disp > 400)
                     mpg cyl disp  hp drat    wt  qsec vs am gear carb
Cadillac Fleetwood  10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
Lincoln Continental 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
Chrysler Imperial   14.7   8  440 230 3.23 5.345 17.42  0  0    3    4

> disp > 400
Error: object 'disp' not found
> disp
Error: object 'disp' not found
```

evaluated inside **mtcars**

- Non-standard evaluation functions don't use the usual lookup rules
- Great for data analysis, because they save typing

Other NSE functions

```
> library(ggplot2)
> ggplot(mpg, aes(displ, cty)) + geom_point()

> library(dplyr)
> filter(mtcars, disp > 400)
  mpg cyl disp  hp drat    wt  qsec vs am gear carb
1 10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
2 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
3 14.7   8  440 230 3.23 5.345 17.42  0  0    3    4

> disp_threshold <- 400
> filter(mtcars, disp > disp_threshold)
   mpg cyl disp  hp drat    wt  qsec vs am gear carb
1 10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
2 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
3 14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
```

Other NSE functions

```
> library(ggplot2)
> ggplot(mpg, aes(displ, cty)) + geom_point()

> library(dplyr)
> filter(mtcars, disp > 400)
  mpg cyl disp  hp drat    wt  qsec vs am gear carb
1 10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
2 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
3 14.7   8  440 230 3.23 5.345 17.42  0  0    3    4

> disp_threshold <- 400
> filter(mtcars, disp > disp_threshold)
   mpg cyl disp  hp drat    wt  qsec vs am gear carb
1 10.4   8  472 205 2.93 5.250 17.98  0  0    3    4
2 10.4   8  460 215 3.00 5.424 17.82  0  0    3    4
3 14.7   8  440 230 3.23 5.345 17.42  0  0    3    4
```

`disp_threshold` value in the global environment

What to do?

- Using non-standard evaluation functions inside your own functions can cause surprises
- Avoid using non-standard evaluation functions inside your functions
- Or, learn the surprising cases and protect against them

R Functions

Let’s practice!

R Functions

Hidden arguments

Pure functions

1. Their output only depends on their inputs
2. They don’t affect the outside world except through their return value

- Hidden arguments are function inputs that may be different for different users or sessions
- Common example: argument defaults that depend on global options

Viewing global options

```
> options()
$add.smooth
[1] TRUE

> options()
$add.smooth
[1] TRUE

$browserNLdisabled
[1] FALSE

$CBoundsCheck
[1] FALSE

$check.bounds
[1] FALSE

...
```

Getting and setting options

```
> getOption("digits")
[1] 7

> options(digits = 5)

> getOption("digits")
[1] 5

# To read about some of the common options
> ?options
```

Relying on options in your code

- The return value of a function should **never** depend on a global option
- Side effects may be controlled by global options

R Functions

Let’s practice!

R Functions

Wrapping up

Writing functions

- If you have copy-and-pasted two times, it's time to write a function
- Solve a simple problem, before writing the function
- A good function is both correct and understandable

Functional Programming

- Abstract away the pattern, so you can focus on the data and actions
- Solve iteration problems more easily
- Have more understandable code

Remove duplication and improve readability

```
> 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$b, 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))
```

```
> library(purrr)
> df[] <- map(df, rescale01)
```

Unusual inputs and outputs

- Deal with failure using `safely()`
- Iterate over two or more arguments
- Iterate functions for their side effects

Write functions that don't surprise

- Use `stop()` and `stopifnot()` to fail early
- Avoid using type-inconsistent functions in your own functions
- Avoid non-standard evaluation functions in your own functions
- Never rely on global options for computational details

Wrapping up

- **Solve the problem that you're working on**
- Never feel bad about using a for loop!
- Get a function that works right, for the easiest 80% of the problem
- In time, you'll learn how to get to 99% with minimal extra effort
- Concise and elegant code is something to strive towards!

Thanks!