Defensive R Programming

DEFENSIVE R PROGRAMMING

Dr. Colin Gillespie
Jumping Rivers

Exported functions

- `library()` gives you direct access to this package box
- `library("dplyr")` gives you direct access to the exported functions in **dplyr**

```
## 238 exported functions
getNamespaceExports("dplyr")
```

- Alternatively, we can use `::` to directly access a function

```
## The filter function from dplyr
dplyr::filter
```

Great minds think alike

- Sometimes package authors come with the same function name

```
## The filter() function
stats::filter()
dplyr::filter()
```

- If I type `filter()` which version do I get?
- It depends on the package load order!

Great minds think alike

- `search()` to the rescue

```
search()
```

```
# [1] ".GlobalEnv"        "package:dplyr"     "package:stats"
# <Other packages>
```

Early warning systems

DEFENSIVE R PROGRAMMING

Dr. Colin Gillespie
Jumping Rivers

Problem 1: TRUE and FALSE

- TRUE and FALSE are special values
- We can't override them

```
TRUE <- 5
```

```
# Error in TRUE <- 5 : invalid (do_set) left-hand side to assignment
```

Problem 2: TRUE and FALSE

Suppose we are working out an F-statistic. It would be natural to have

```
# df is the F-density function
(F <- df(1, 9, 67))
```

```
[1] 0.7798
```

But R treats positive numbers as `TRUE`, so

```
if(F) message("Yer aff yer heid!")
```

```
Yer aff yer heid!
```

`F` is now treated as `TRUE`!

- Get in the habit of using `TRUE` and `FALSE`
  - not `T` and `F`
    - If you testing for `TRUE`, use `isTRUE()`

```
isTRUE(T)
```

```
[1] TRUE
```

```
isTRUE(2)
```

```
[1] FALSE
```

```
T <- 10
isTRUE(T)
```

```
[1] FALSE
```

We can turn it off with `suppressMessages()`

```
noisy = function(a, b) {
   message("I'm doing stuff")
   a + b
}
noisy(1, 2)
```

```
I'm doing stuff
# [1] 3
```

```
suppressMessages(noisy(1, 2))
```

```
# [1] 3
```

Telling packages to be quiet

- Occasionally, packages can be a bit noisy
- Sometimes loading **ggplot2**, it presents a message
- Don't worry, we can tell it to be quiet

```
suppressPackageStartupMessages(library("ggplot2"))
```

Using message()

The `message()` function is helpful for letting

- you
- and other users

know what's happening.

It's very handy for long running processes

The warning message

The `warning()` function

```
warning("You have been warned!")
```

```
# Warning message:
# You have been warned!
```

- signals that something may have gone wrong
- R continues (unlike an error)
- "Warning message:" is (pre) appended

Suppress Warnings

Similar to messages, you can suppress warnings via

- `suppressWarnings()`

This is almost never a good idea

- Fix the underlying problem!

A good use of warning

Suppose we're performing regression on

```
d = data.frame(y = 1:4, x1 = 1:4)
d$x2 = d$x1 + 1
```

So `x2 = x1 + 1`

When we fit a multiple linear regression model

```
m = lm(y ~ x1 + x2, data = d)
summary(m)
```

```
# Some output removed
# Warning message:
# In summary.lm(m) : essentially perfect fit: summary may be unreliable
```

I saw the sign

Sometimes things are just broken

- We need to raise an error

For example:

```
1 + "stuff"
```

```
Error in 1 + "stuff": non-numeric argument to binary operator
```

Stop right now thank you very much

To raise an error, we use the `stop()` function

```
stop("A Euro 1996 error has occurred")
```

```
# Error: A Euro 1996 error has occurred
```

```
conf_int <- function(mean, std_dev) {
   if(std_dev <= 0)
       stop("Standard deviation must be positive")

   c(mean - 1.96 * std_dev, mean - 1.96 * std_dev)
}
```

The try() function

The `try()` function acts a bit like `suppress()`

```
res <- try("Scotland" + "World cup", silent = TRUE)
```

It tries to execute something, if it doesn't work, it moves on

The try() idiom

```
res <- try("Scotland" + "World cup", silent = TRUE)
res
```

```
[1] 'Error in "Scotland" + "World cup": non-numeric argument to binary operator'
attr(,"class")
[1] "try-error"
attr(,"condition")
<simpleError in "Scotland" + "World cup":
                            non-numeric argument to binary operator>
```

The try() idiom

```
result <- try("Scotland" + "World cup", silent = TRUE)
class(result)
```

```
[1] "try-error"
```

```
if(class(result) == "try-error")
  ## Do something useful
```

Preparing your defences

DEFENSIVE R PROGRAMMING

Dr. Colin Gillespie
Jumping Rivers

datacamp

Tip 1: Avoid obvious comments

- What's obvious is sometimes hard to decide
  - For example, the comments

```
# Loop through data sets
for (dataset in datasets) {
 # Read in data set
 r <- read.csv(dataset)
}
```

    look reasonable

  - But are perhaps a little too obvious

Tip 2: Avoid comments that you will never update

The most common example is header comments at the top of the file

```
# Last updated: 1967-02-25
# Author: D Law
# Status: No 1
```

- These sorts of comments are almost never updated
- I once saw `# list of packages used: XXX, YYY`

Tip 3: Be consistent

- Always start with a single `#` or double `##`
- Start with a capital letter - follow the rules of grammar
- Be careful with jokes
  - What you find funny, others may take offense
- Be sure to comment on code that "looks wrong"
- Use **# TODO** or **# XXX** to indicate a future problem

The full stop

In R, the full stop has a very special meaning

- It is the mechanism that is used in S3 OOP
- When you call the `summary()` function
  - R looks for the function `summary.class_name`

Example: the summary() function

So when you call

```
summary(m)
```

you end up calling

```
summary.lm(m)
```

The **key** point here, is that the full stop is very important

One bit of advice

- There are few R rules that everyone agrees on
- But **everyone** agrees that you should avoid `.` in variable names
- It just prevents confusion

Coding Style

DEFENSIVE R PROGRAMMING

Dr. Colin Gillespie
Jumping Rivers

Uncontroversial rules

- Assignment wars: `=` vs `->`

```
x = 5
# or
x <- 5
```

- Everyone agrees you shouldn't mix & match
- I prefer the superior `=` for assignment but
- DataCamp prefers `<-` for their courses

So be **consistent**

Spacing

Two widely accepted rules are

- spaces around assignment `x <- 5`
- spaces after a comma - `x[1, 1]` instead of `x[1,1]`

lintr in Action

Suppose I have the following code

```
my_bad<-function(x, y) {
  x+y
}
```

saved in the file `code.R`.

- Running `lint::lintr("code.R")` highlights two issues

Issue 1

```
my_bad<-function(x,y) {
  x+y
}
```

```
r[[1]]
tmp.R:1:7: style: Put spaces around all infix operators.
my_bad<-function(x,y) {
     ~^~~
```

```
my_bad <- function()
```

Issue 2

```
my_bad<-function(x,y) {
  x+y
}
```

```
r[[3]]
tmp.R:2:4: style: Put spaces around all infix operators.
  x+y
  ~^~
```

```
 x + y
```

Organizing a project

DEFENSIVE R PROGRAMMING

Dr. Colin Gillespie
Jumping Rivers

Project Set-up

Every project I work on

- Has its own directory
- Has a sensible name

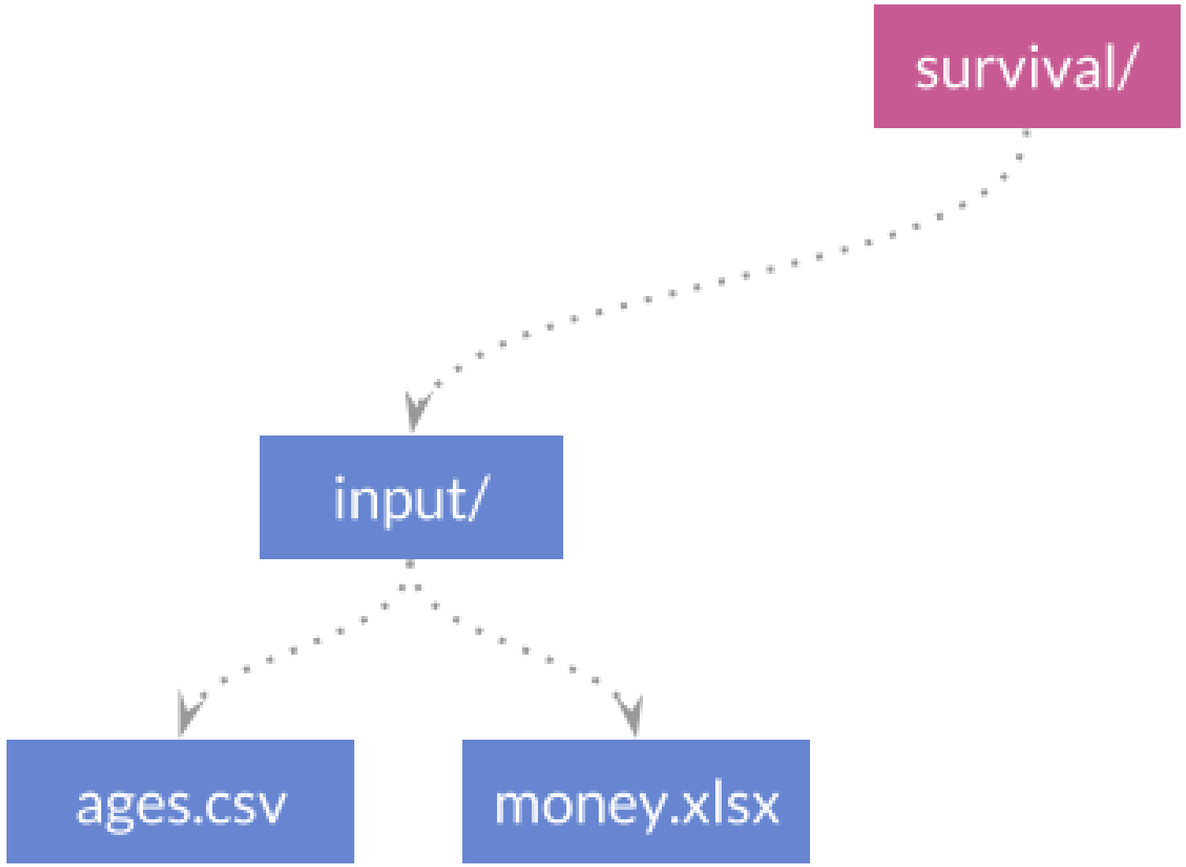
The directory name gives the **context** of the scripts

Directory: input/

This directory contains data, typically

- `csv` & excel files
- No R code

Data is only edited in R



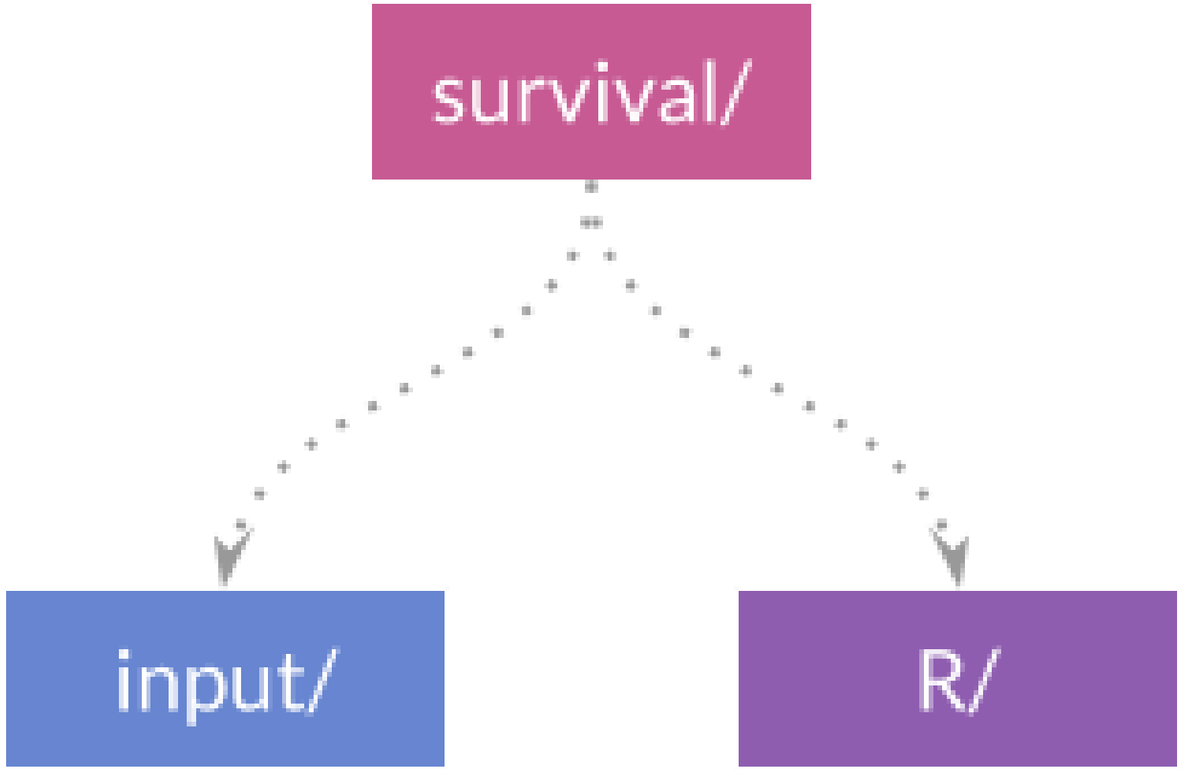
Directory: R/

All R code lives in this directory

Notice The directory isn't

- R_analysis
- R_code
- R_survival

just plain **R/**



Directory: R/

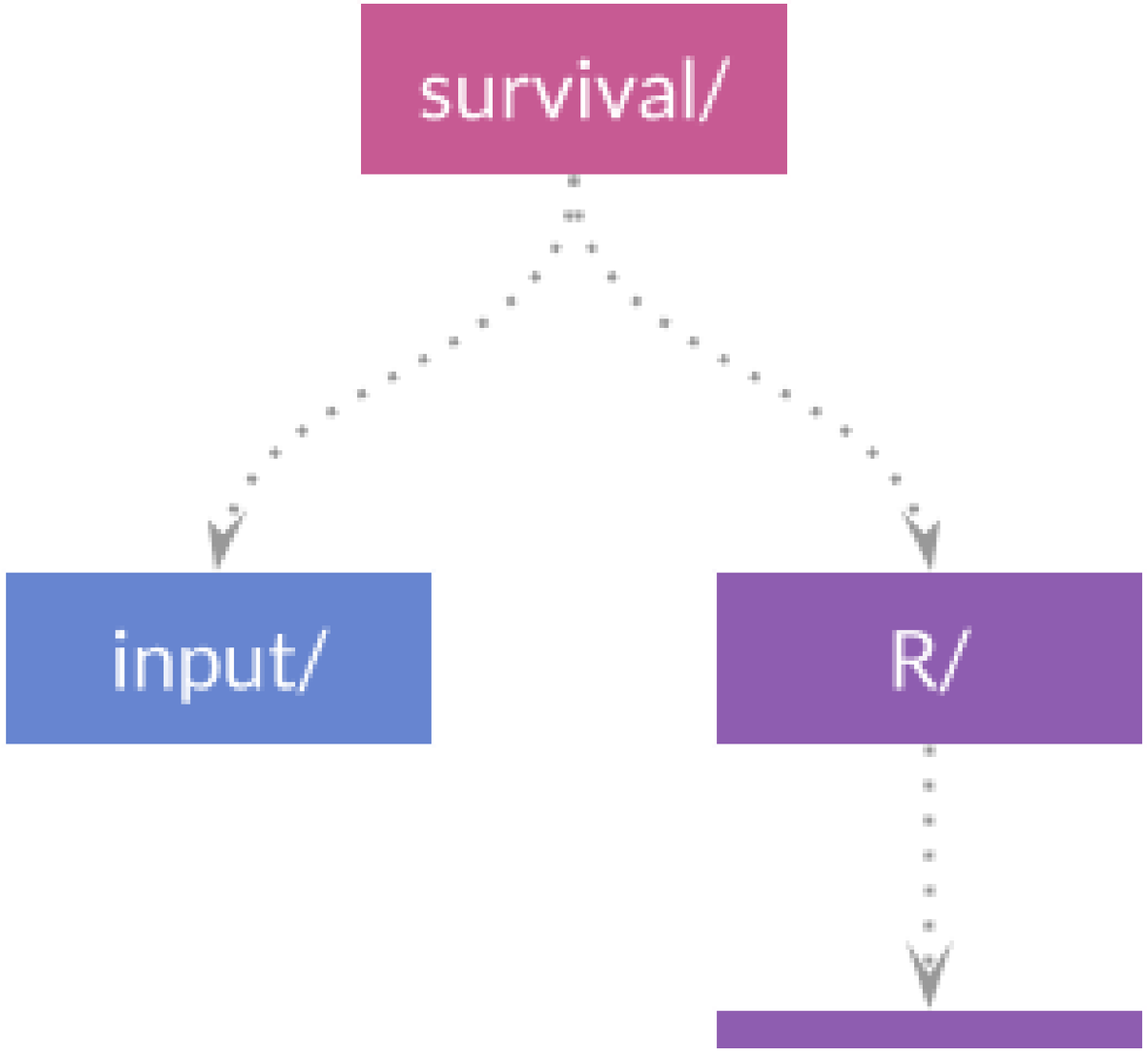
In this directory, I always have a file called

- `load.R`

This file loads the data from `input/`

Every project I've worked has a similar structure

- I can give you any project and you can load the data

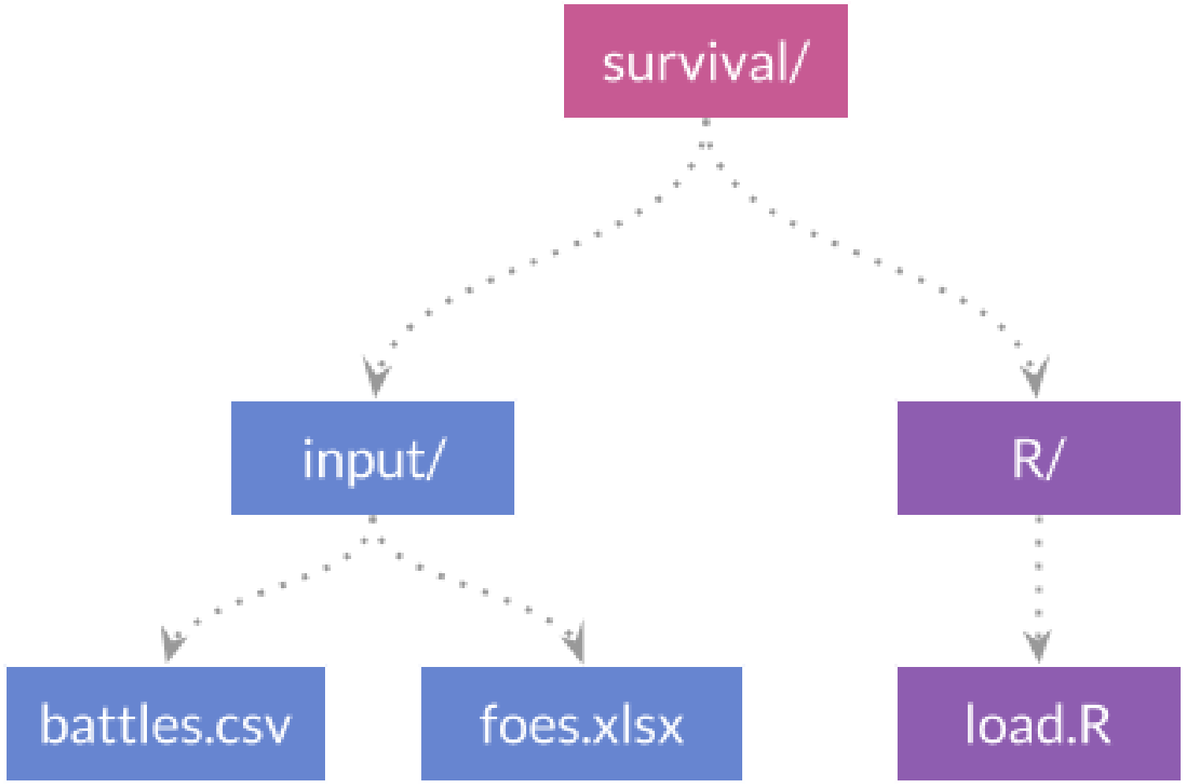


The load.R file

All paths are relative

```
battles <- read_csv("input/battles.csv")
foes <- read_xlsx("input/foes.xlsx")
```

My code is portable



Other R files

Remember, all R files live in the R directory!

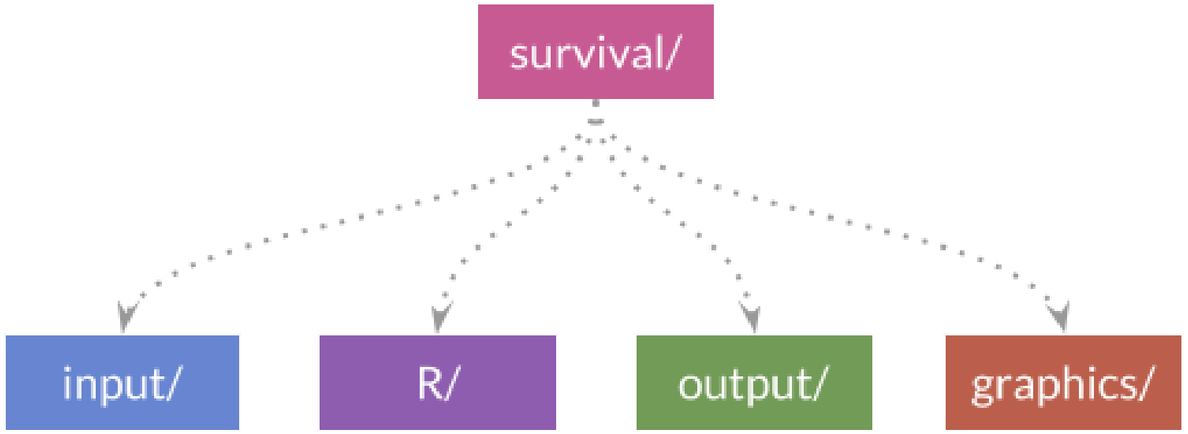
- `clean.R` - for cleaning your data
- `function.R` - any helper functions
- `analysis.R` - the actual analysis

Standard names used in every project

Project overview 5

So far we have encountered

- the base project directory
- `R/` for R scripts
- `input/` for data sets



In this last video, we'll look at

- `output/` for output **generated** data files
- `graphics/` for **generated** plots

The graphics/ directory

This directory just contains

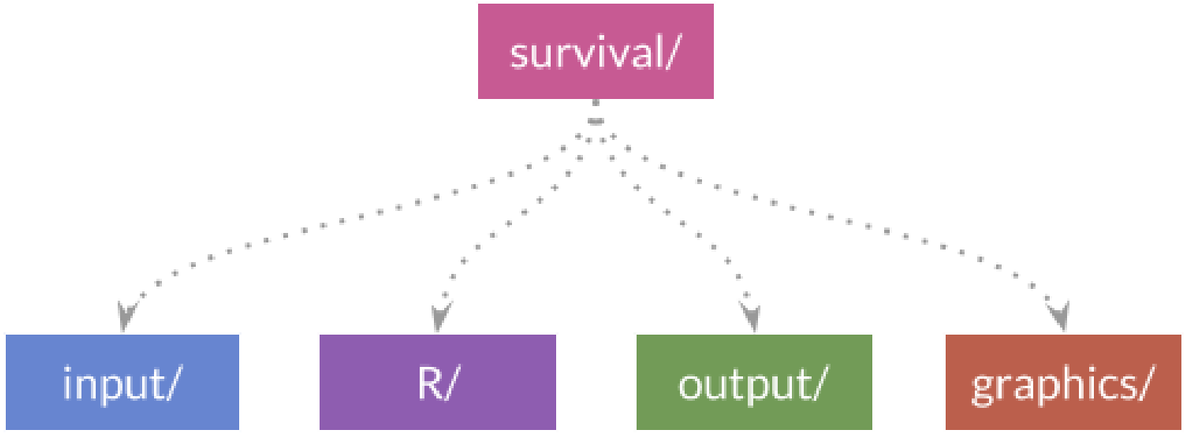
- **graphics**!

In my `R/` directory I have imaginatively named script

- `graphics.R`

that generates all graphics

Make sure to use *relative* paths!



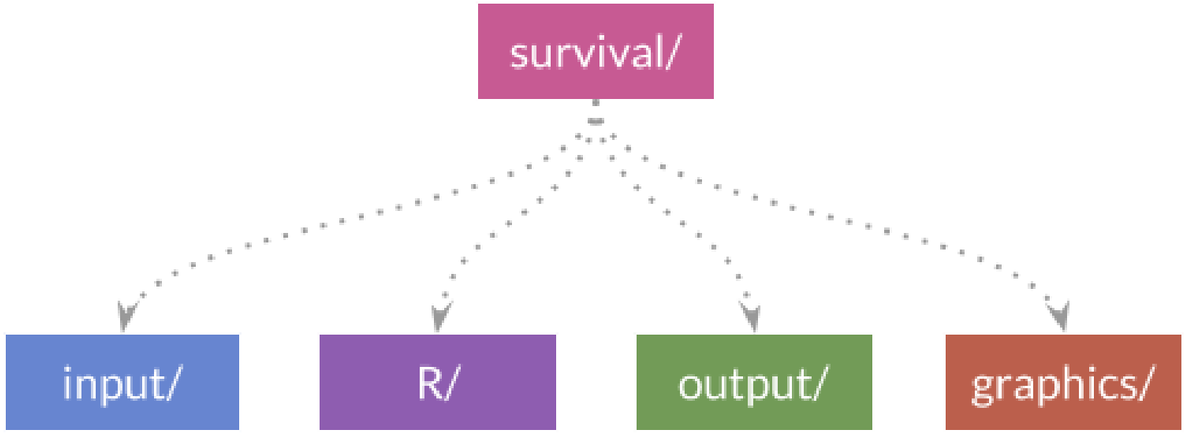
The output/ directory

This directory contains **output**

For example

- List of significant variables, perhaps p-value
- Data for the next analysis

Personally, I typically don't use this directory



A battle plan

DEFENSIVE R PROGRAMMING

Dr. Colin Gillespie
Jumping Rivers

Multiple words

Filenames often contain multiple words

For example

- `cluster-analysis.R`
- `load-survival-data.R`
- `plot-residuals.R`

We can learn from slugs

- Use sensible names
  - `ac.R` or `analysis-clustering.R`
  - `1.R` or `loading.R`
    - Be consistent
    - Use the same file extension - `.R`
    - Always lower case

Dates - ISO8601

Dates should be

> YYYY-MM-DD

- All dates are now in an obvious and natural order
- Sorting just works!

```
2017-01-02
2018-01-01
2018-01-02
```

Numbers are good

For this course, I created directories called

- `chapter01`
- `chapter02`

Simple, yet effective

Code Smells and Feels

rstd.io/code-smells

Jennifer Bryan
RStudio

@jennybc
@JennyBryan

refactor

make code

- easier to understand
- cheaper to modify

without changing behaviour

"Never use `attach()`."

"Always put a space before and after `=`."

"Have better taste."

"Write more elegant code."

x	bizarro(x)
-77, 0, 4	77, 0, -4
TRUE, FALSE	FALSE, TRUE
"desserts", "god"	"stressed", "dog"

Tip #1:
Do not comment and uncomment sections of code to alter behaviour.

Tip #1a:
Especially not in multiple places that you will never, ever keep track of 😉

Tip #2:
Use `if () else ()` in moderation.

Tip #2a:
Describe in grandiose terms:
"I used a one layer neural network with identity activation and no hidden layers." *-- Federico Vaggi*

Tip #3:
Use functions.

Tip #3a:
A few little functions >> a monster function.

Tip #3b:
Small well-named helper >> commented code

```r
bizarro <- function(x) {
  if (is.numeric(x)) {
    -x
  } else if (is.logical(x)) {
    !x
  } else { stop(...) }
}

bizarro(1:5)
#> [1] -1 -2 -3 -4 -5

bizarro(c("abc", "def"))
#> Error: Don't know how to make bizzaro <character>
```

Tip #4:
Use proper functions for handling class & type.
Use simple conditions.

Extract fussy condition logic into well-named function.

from googledrive

```
drive_cp <- function(file, ...) {
  ...
  if (is_parental(file)) {
    stop_glue("The Drive API does not copy folders or Team Drives.")
  }
  ...
}
```

```
is_parental <- function(d) {
  stopifnot(inherits(d, "dribble"))
  kind <- purrr::map_chr(d$drive_resource, "kind")
  mime_type <- purrr::map_chr(d$drive_resource, "mimeType", .default = NA)
  kind == "drive#teamDrive" | mime_type == "application/vnd.google-apps.folder"
}
```

Tip #5:

Hoist quick `stop()`s and `return()`s to the top.

I.e., use a guard clause.

Clarify and emphasize the happy path.

```r
bizarro <- function(x) {
  stopifnot(is.numeric(x) || is.logical(x))

  if (is.numeric(x)) {
    -x
  } else {
    !x
  }
}

bizarro(c(TRUE, FALSE, FALSE, TRUE, FALSE))
#> [1] FALSE  TRUE  TRUE FALSE  TRUE

bizarro(1:5)
#> [1] -1 -2 -3 -4 -5

bizarro(c("abc", "def"))
#> Error: is.numeric(x) || is.logical(x) is not TRUE
```

```
get_some_data <- function(config, outfile) {
  if (config_bad(config)) {
    stop("Bad config")
  }

  if (!can_write(outfile)) {
    stop("Can't write outfile")
  }

  if (!can_open_network_connection(config)) {
    stop("Can't access network")
  }

  data <- parse_something_from_network()
  if(!makes_sense(data)) {
    return(FALSE)
  }

  data <- beautify(data)
  write_it(data, outfile)
  TRUE
}
```

```
get_some_data <- function(config, outfile) {
  if (config_ok(config)) {
    if (can_write(outfile)) {
      if (can_open_network_connection(config)) {
        data <- parse_something_from_network()
        if(makes_sense(data)) {
          data <- beautify(data)
          write_it(data, outfile)
          return(TRUE)
        } else {
          return(FALSE)
        }
      } else {
        stop("Can't access network")
      }
    } else {
      ## uhm. What was this else for again?
    }
  } else {
    ## maybe, some bad news about ... the config?
  }
}
```

```
get_some_data <- function(config, outfile) {
  if (config_bad(config)) {
    stop("Bad config")
  }

  if (!can_write(outfile)) {
    stop("Can't write outfile")
  }

  if (!can_open_network_connection(config)) {
    stop("Can't access network")
  }

  data <- parse_something_from_network()
  if(!makes_sense(data)) {
    return(FALSE)
  }

  data <- beautify(data)
  write_it(data, outfile)
  TRUE
}
```

Tip #5c:

An `if` block that ends with `stop()` or `return()` does not require `else`.

Recognize your early exits.

less indentation >> lots of indentation

Tip #6:

If your conditions deal with class, it's time to get object-oriented (OO).

In CS jargon, use polymorphism.

```r
bizarro <- function(x) {
  if (is.numeric(x)) {
    -x
  } else if (is.logical(x)) {
    !x
  } else if (is.character(x)) {
    str_reverse(x)
  } else if (is.factor(x)) {
    levels(x) <- rev(levels(x))
    x
  } else {
    stop(...)
  }
}
```

S3 OO system

```
bizarro <- function(x) {
  UseMethod("bizarro")
}

bizarro.default <- function(x) {
  stop(
    "Don't know how to make bizzaro <",
    class(x)[[1]], ">",
    call. = FALSE
  )
}
```

```
bizarro(1:5)
#> Error: Don't know how to make bizzaro <integer>

bizarro(TRUE)
#> Error: Don't know how to make bizzaro <logical>

bizarro("abc")
#> Error: Don't know how to make bizzaro <character>
```

S3 OO system

```
bizarro.numeric <- function(x) -x

bizarro.logical <- function(x) !x

bizarro.character <- function(x) str_reverse(x)

bizarro.factor <- function(x) {
  levels(x) <- rev(levels(x))
  x
}

bizarro.data.frame <- function(x) {
  names(x) <- bizarro(names(x))
  x[] <- lapply(x, bizarro)
  x
}
```

```r
str_reverse <- function(x) {
  vapply(
    strsplit(x, ""),
    FUN = function(z) paste(rev(z), collapse = ""),
    FUN.VALUE = ""
  )
}

str_reverse(c("abc", "def"))
#> [1] "cba" "fed"
```

```
bizarro(1:5)
#> [1] -1 -2 -3 -4 -5

bizarro(c(TRUE, FALSE, FALSE, TRUE, FALSE))
#> [1] FALSE  TRUE  TRUE FALSE  TRUE

bizarro(c("abc", "def"))
#> [1] "cba" "fed"

(m <- factor(month.abb[1:3], levels = month.abb[1:3]))
#> [1] Jan Feb Mar
#> Levels: Jan Feb Mar
bizarro(m)
#> [1] Mar Feb Jan
#> Levels: Mar Feb Jan

bizarro(head(iris, 3))
#>   htgneL.lapeS htdiW.lapeS htgneL.lateP htdiW.lateP   seicepS
#> 1          -5.1         -3.5          -1.4         -0.2 virginica
#> 2          -4.9         -3.0          -1.4         -0.2 virginica
#> 3          -4.7         -3.2          -1.3         -0.2 virginica
```

Tip #6 redux:

Avoid explicit conditionals by creating methods that are specific to an object's class.

from stringr

```
str_pad <- function(string,
                    width,
                    side = c("left", "right", "both"),
                    pad = " ") {
  side <- match.arg(side)

  switch(
    side,
    left  =  stri_pad_left(string, width, pad = pad),
    right = stri_pad_right(string, width, pad = pad),
    both  =  stri_pad_both(string, width, pad = pad)
  )
}
```

Tip #7:
`switch()` is ideal if you need to dispatch different logic, based on a string.

Tip #7a: You are allowed to write a helper function to generate that string.

```
library(tidyverse)

tibble(
  age_yrs = c(0, 4, 10, 15, 24, 55),
  age_cat = case_when(
    age_yrs < 2  ~ "baby",
    age_yrs < 13 ~ "kid",
    age_yrs < 20 ~ "teen",
    TRUE         ~ "adult"
  )
)
#> # A tibble: 6 x 2
#>   age_yrs age_cat
#>     <dbl> <chr>
#> 1       0 baby
#> 2       4 kid
#> 3      10 kid
#> 4      15 teen
#> 5      24 adult
#> 6      55 adult
```

alternative to:

```
ifelse(age_yrs < 2, "baby",
       ifelse(age_yrs < 13, "kid",
              ifelse(age_yrs < 20, "teen",
                     "adult"
              )
       )
)
```

Tip #8:

`dplyr::case_when()` is ideal if you need to dispatch different data, based on data (+ logic).

from rlang, purrr

```r
`%||%` <- function(x, y) {
  if (is_null(x)) y else x
}
```

from devtools

```
github_remote <- function(repo, username = NULL, ...) {
  meta <- parse_git_repo(repo)
  ...
  meta$username <- username %||%
    getOption("github.user") %||%
    stop("Unknown username.")
  ...
}
```

Write simple conditions.

Use helper functions.

Handle class properly.

Return and exit early.

polymorphism

`switch()`

`case_when()`

`%||%`

rstd.io/code-smells

@jennybc @JennyBryan