GG606

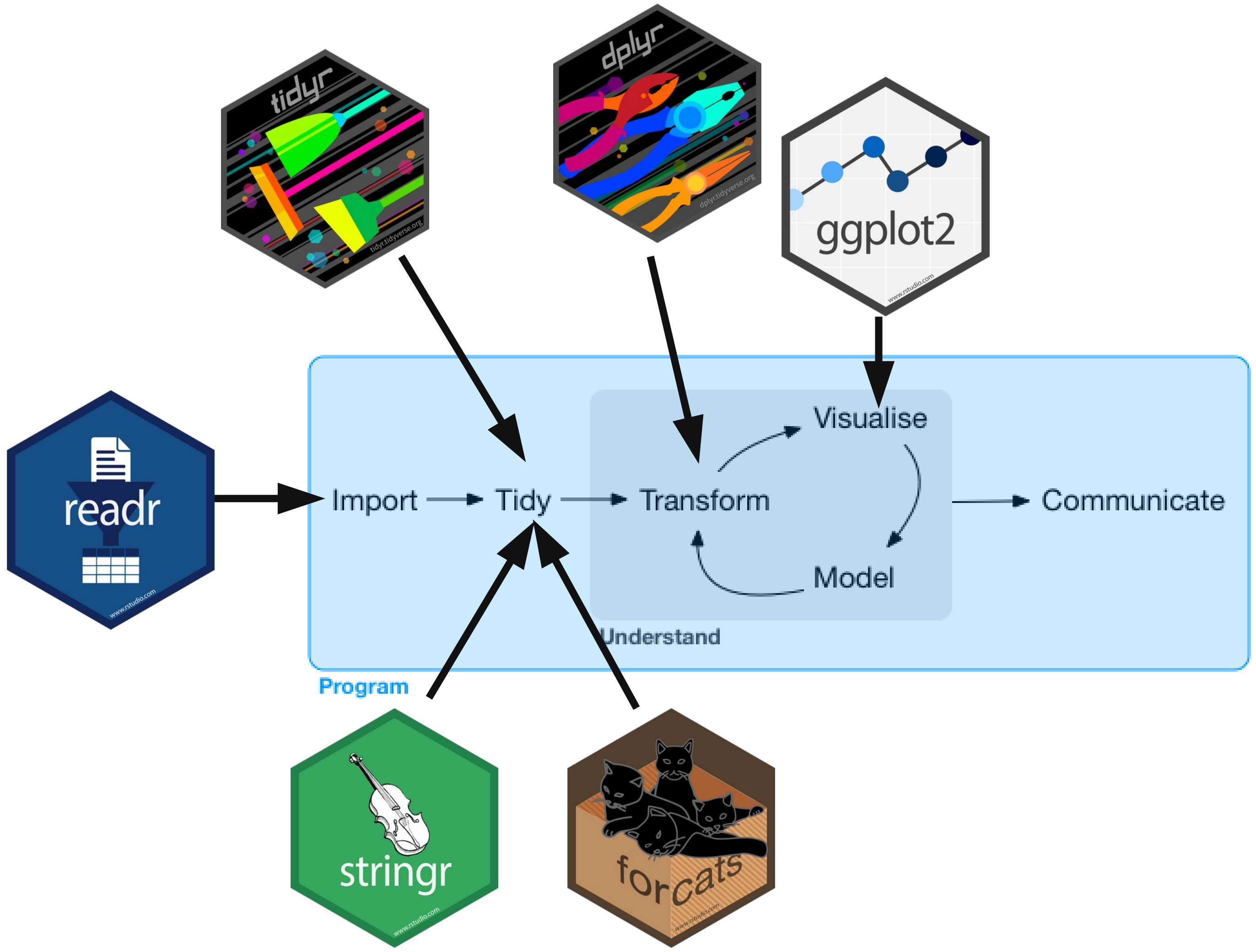
Functions & Packages

Homework

- Functions for variance and skewness
- Function to load the penguins.csv
  do a calculation, and make a figure
- Call the function like this:
  `my_fancy_function(filename)`

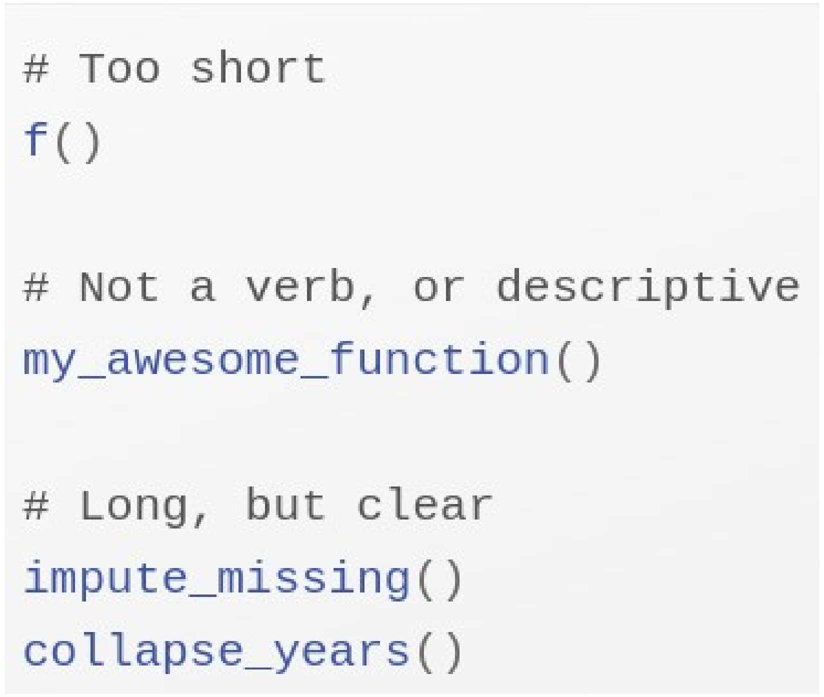
$$\mathrm{Var}(x) = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

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


tidyr
dplyr
ggplot2
readr
stringr
forcats
Import
Tidy
Transform
Visualise
Model
Communicate
Understand
Program

Functions for humans & computers

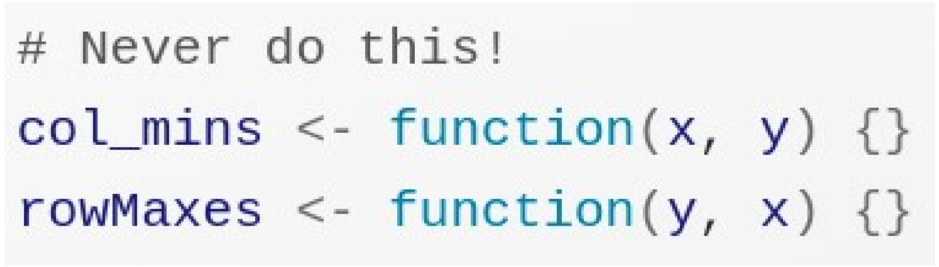
- verb(nouns)



```r
# Too short
f()

# Not a verb, or descriptive
my_awesome_function()

# Long, but clear
impute_missing()
collapse_years()
```

Functions for humans & computers

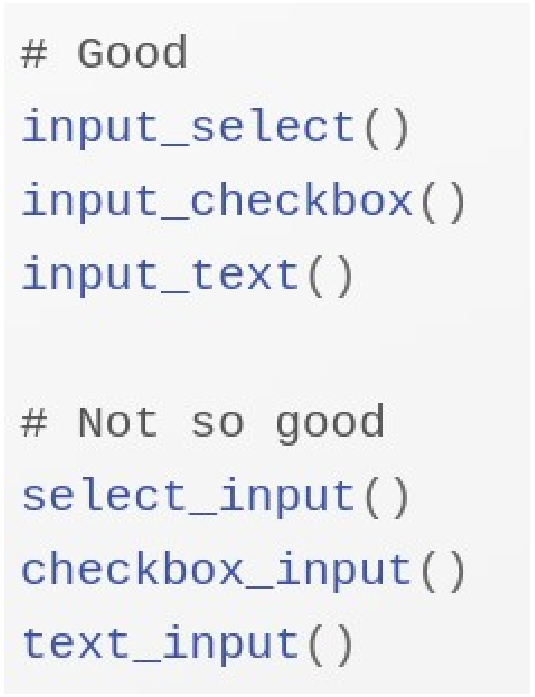
- So much wrong!
- *snake_case*
- *camelCase*



```
# Never do this!
col_mins <- function(x, y) {}
rowMaxes <- function(y, x) {}
```

Functions for humans & computers

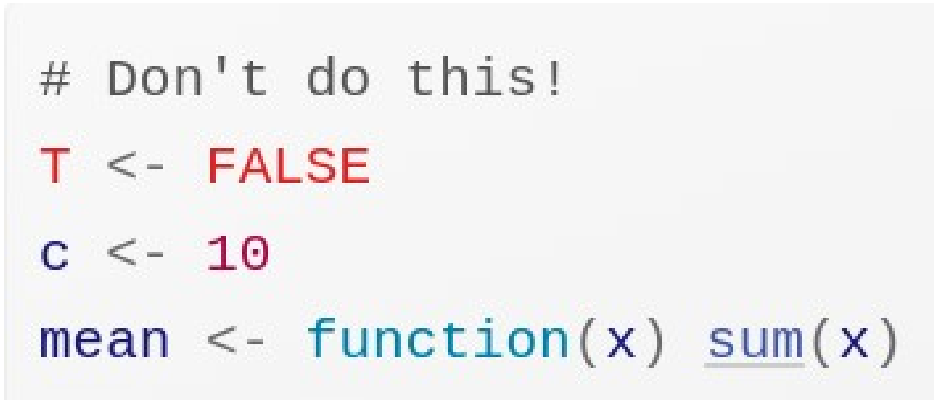
- Common prefix
- `stringr::str_`
- Avoid overwriting



```
# Good
input_select()
input_checkbox()
input_text()

# Not so good
select_input()
checkbox_input()
text_input()
```

Functions for humans & computers

- Hilarious joke to play on your friends



```r
# Don't do this!
T <- FALSE
c <- 10
mean <- function(x) sum(x)
```

Puzzle

-
  ```
  f1 ← function(string, prefix) {
    substr(string, 1, nchar(prefix)) == prefix
  }
  ```

-
  ```
  f2 ← function(x) {
    if (length(x) ⩽ 1) return(NULL)
    x[-length(x)]
  }
  ```

-
  ```
  f3 ← function(x, y) {
    rep(y, length.out = length(x))
  }
  ```

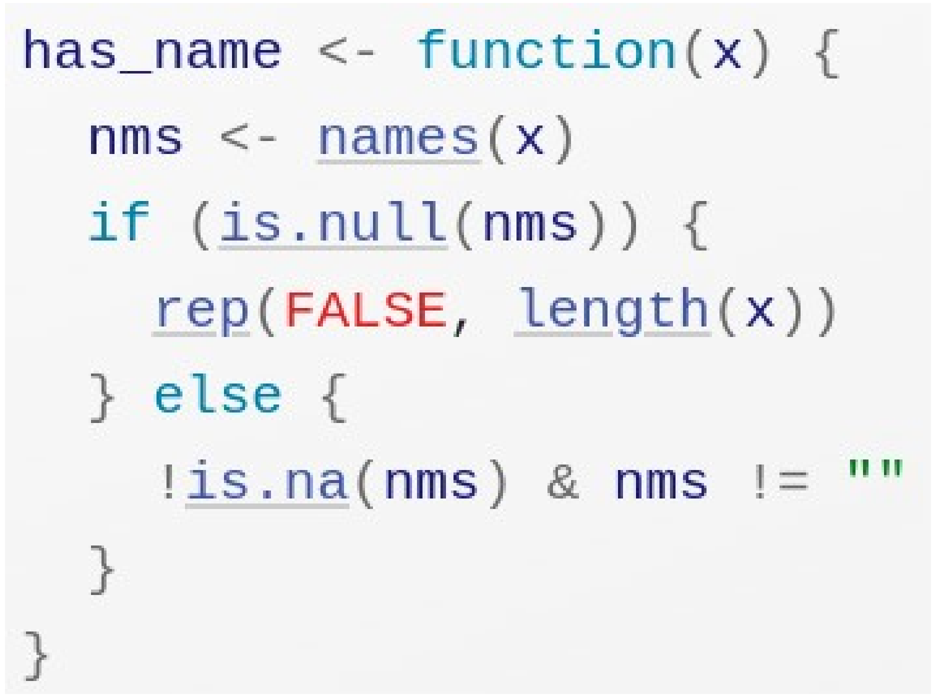
Conditions

- R: if() elseif() else
- Python: if : else:
- C++: if() elseif() else
- Matlab: if elseif else end
- Excel: IF(a,b,c)

How many have experience with if statements?

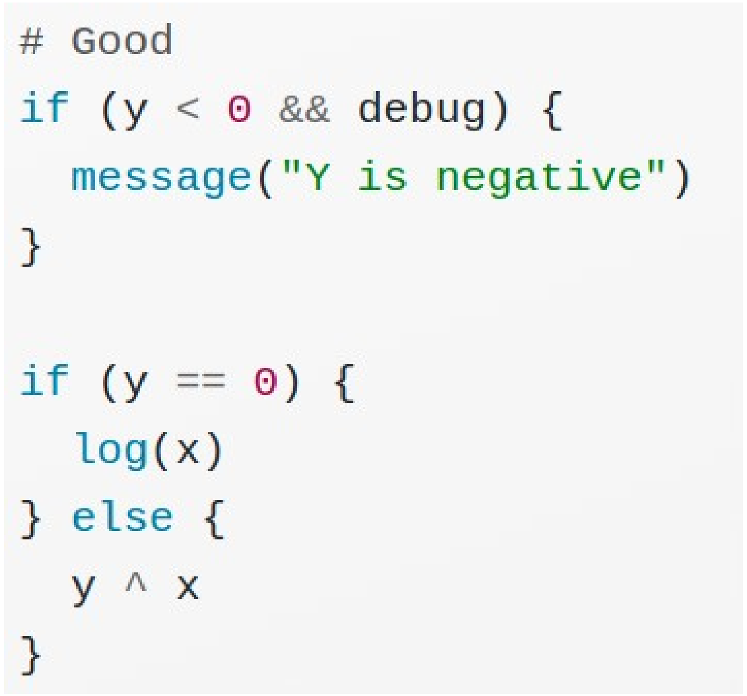
Simple Example

- What does it do
- How are `if` and `else` working?
- What will output(s) look like?

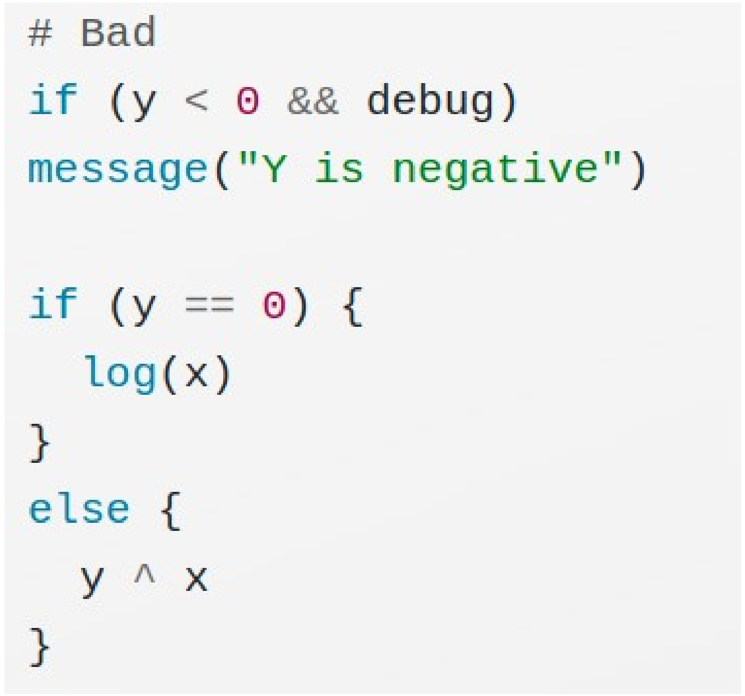


```r
has_name <- function(x) {
  nms <- names(x)
  if (is.null(nms)) {
    rep(FALSE, length(x))
  } else {
    !is.na(nms) & nms != ""
  }
}
```

Code Style



```r
# Good
if (y < 0 && debug) {
  message("Y is negative")
}

if (y == 0) {
  log(x)
} else {
  y ^ x
}
```



```r
# Bad
if (y < 0 && debug)
message("Y is negative")

if (y == 0) {
  log(x)
}
else {
  y ^ x
}
```

Try These

- Write a greeting function that says “good morning”, “good afternoon”, or “good evening”, depending on the time of day. (Hint: use a time argument that defaults to `lubridate::now()`. That will make it easier to test your function.)
- Implement a `fizzbuzz` function. It takes a single number as input. If the number is divisible by three, it returns “fizz”. If it’s divisible by five it returns “buzz”. If it’s divisible by three and five, it returns “fizzbuzz”. Otherwise, it returns the number. Make sure you first write working code before you create the function. (Hint: the modulo operator %% will be useful.)

Function arguments

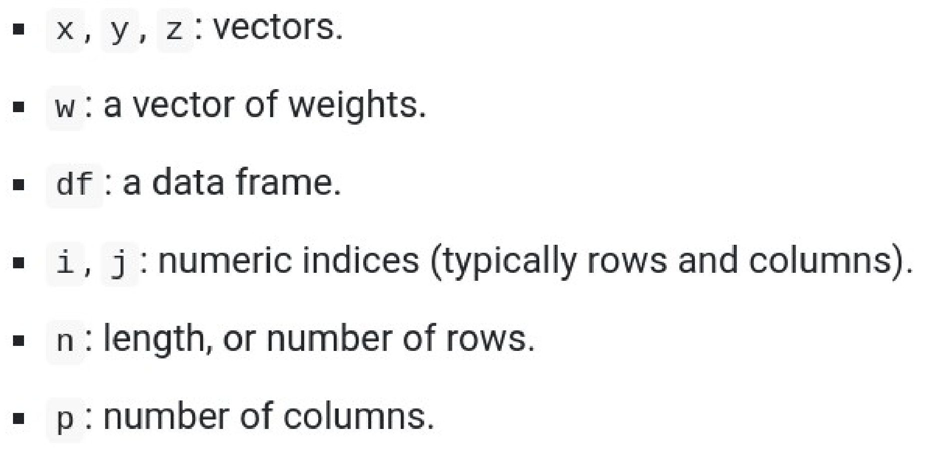
- Data first then details
- `log(x =, base = )`
- `mean(x = , trim = , na.rm = )`
- `t.test(x = )`

Type `log(` and hit TAB

White space is your friend

Argument names

- Longer & descriptive
- Some common short names
- Match existing arguments



- `x`, `y`, `z`: vectors.
- `w`: a vector of weights.
- `df`: a data frame.
- `i`, `j`: numeric indices (typically rows and columns).
- `n`: length, or number of rows.
- `p`: number of columns.

`df` is not a good idea

Check some|all conditions

```r
wt_mean <- function(x, w) {
  sum(x * w) / sum(w)
}
```

```r
wt_mean <- function(x, w) {
  if (length(x) != length(w)) {
    stop("`x` and `w` must be the same length", call. = FALSE)
  }
  sum(w * x) / sum(w)
}
```

`?stopifnot`

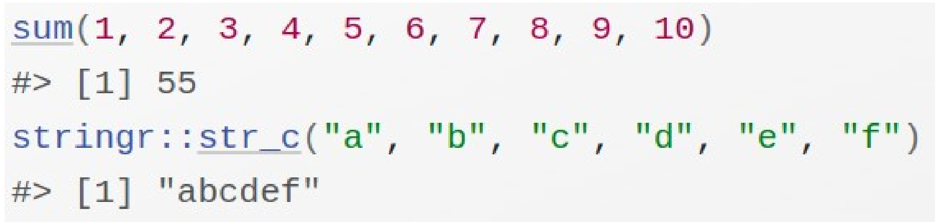
Ensure the Truth of R Expressions

Weighted mean

R’s recycling rules

Dot-dot-dot (...)

- Arbitrary number of inputs
- ... captures them and makes them available to other functions



```
sum(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
#> [1] 55
stringr::str_c("a", "b", "c", "d", "e", "f")
#> [1] "abcdef"
```

Examples?

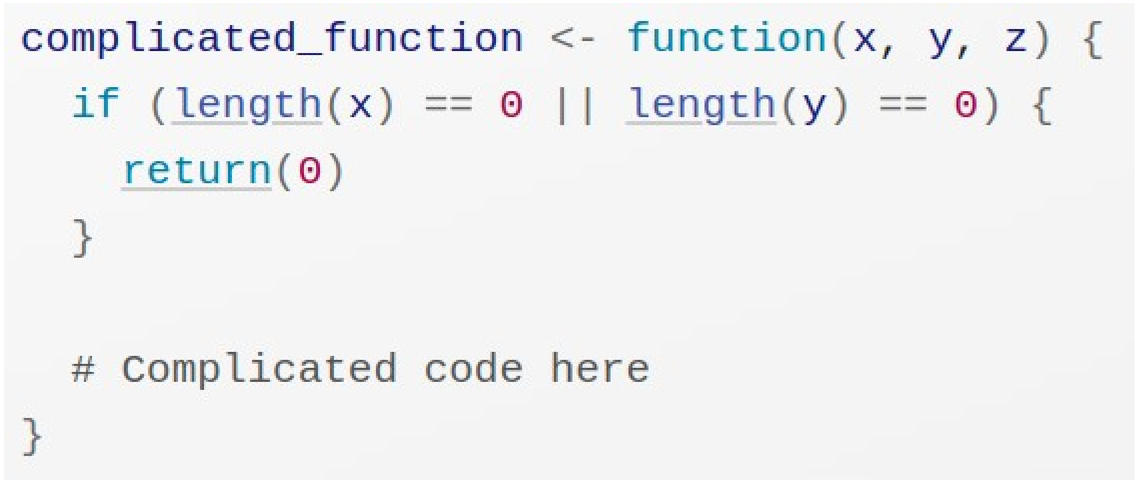
Dot-dot-dot (...)

```
commas <- function(...) stringr::str_c(..., collapse = ", ")
commas(letters[1:10])
#> [1] "a, b, c, d, e, f, g, h, i, j"
```

Simplicity

Returns

- Functions returns something
- Return the last evaluation
- Can use `return()`, but when?



```r
complicated_function <- function(x, y, z) {
  if (length(x) == 0 || length(y) == 0) {
    return(0)
  }

  # Complicated code here
}
```

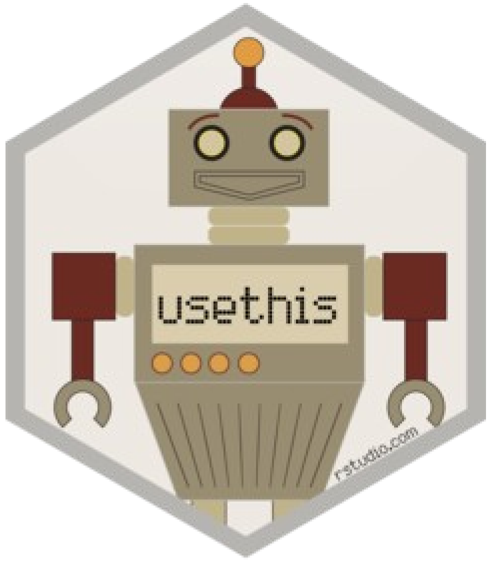
Returns

```r
f <- function() {
  if (x) {
    # Do
    # something
    # that
    # takes
    # many
    # lines
    # to
    # express
  } else {
    # return something short
  }
}
```

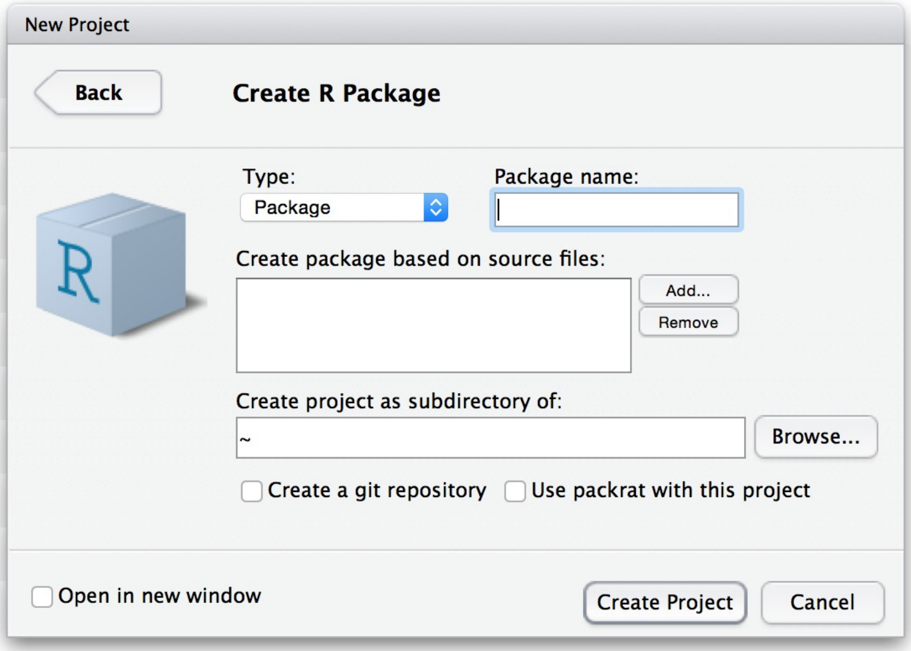
```r
f <- function() {
  if (!x) {
    return(something_short)
  }

  # Do
  # something
  # that
  # takes
  # many
  # lines
  # to
  # express
}
```

Packages

- Many ways to do this
  - Two simple ways: RStudio & `usethis`
- Walk through both
- Very few *required* parts to a package



RStudio



Type: Package

Package name:

Create package based on source files:

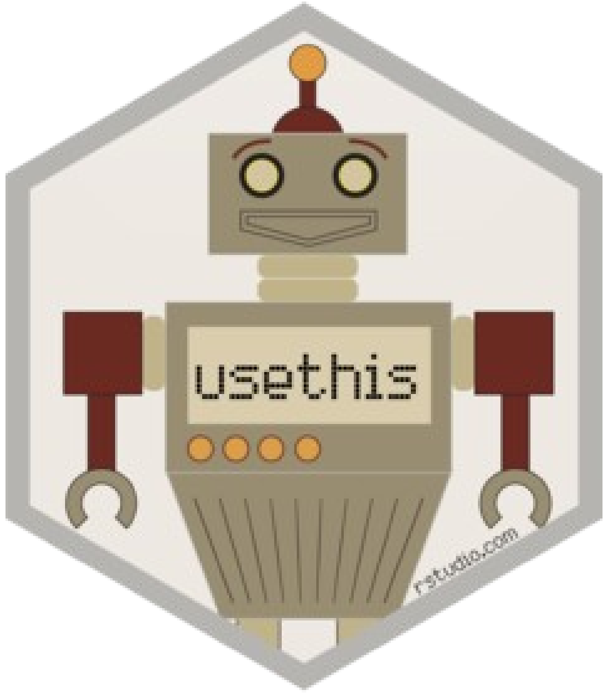
Create project as subdirectory of: ~

☐ Create a git repository ☐ Use packrat with this project

☐ Open in new window

- Project
- New Project
- R Package
- Minimal contents appear

usethis



- create_package()
- use_r()
- loadall()
- check()
- document()