Advanced Function Writing

Justin Post

Recap!

- Function writing opens R up!
- Syntax

```
nameOfFunction <- function(input1, input2, ...) {
  #code
  #return something with return()
  #or returns last value
}
```

- Can set defaults in function definition
- Can return a named list
- Can give unnamed arguments for use

Going Further

A few more useful topics:

- `...` for unnamed arguments
- Writing `tidyverse` style functions
- Lazy evaluation
- Environments and lexical scoping

Unnamed Arguments

- Sometimes we want to
  - supply arguments to functions used in the body of our function
  - allow the user to specify more than one argument (say column)
- Consider the first argument of `data.frame()`

```
data.frame
```

```
## function (..., row.names = NULL, check.rows = FALSE, check.names = TRUE,
##     fix.empty.names = TRUE, stringsAsFactors = FALSE)
## {
##     data.row.names <- if (check.rows && is.null(row.names))
##         function(current, new, i) {
##             if (is.character(current))
##                 new <- as.character(new)
##             if (is.character(new))
##                 current <- as.character(current)
##             if (anyDuplicated(new))
##                 return(current)
##             if (is.null(current))
##                 
##                                     new) || all(current == ""))
##                 
##             stop(gettextf("mismatch of row names in arguments of 'data.frame', item %d",
```

Our `standardize()` Function

Recall the function we wrote a while back:

```
standardize <- function(vector, center = TRUE, scale = TRUE) {
  mean <- mean(vector)
  stdev <- sd(vector)
  if (center) {
    vector <- vector - mean
    }
  if (scale) {
    vector <- vector / stdev
    }
  return(list(result = vector, mean = mean, sd = stdev))
  }
```

Unnamed Arguments

- Add unnamed arguments to our function for use with `sd()` and `mean()`

```
 sd
```

```
## function (x, na.rm = FALSE)
## sqrt(var(if (is.vector(x) || is.factor(x)) x else as.double(x),
##     na.rm = na.rm))
## <bytecode: 0x00000000210e23d8>
## <environment: namespace:stats>
```

```
 mean.default
```

```
## function (x, trim = 0, na.rm = FALSE, ...)
## {
##     if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
##         warning("argument is not numeric or logical: returning NA")
##         return(NA_real_)
##     }
##     if (na.rm)
##         x <- x[!is.na(x)]
##     if (!is.numeric(trim) || length(trim) != 1L)
##         stop("'trim' must be numeric of length one")
##     n <- length(x)
##     if (trim > 0.0 ) {
##         if (is.complex(x))
##             stop("trimmed means are not defined for complex data")
##         if (anyNA(x))
##             return(NA_real_)
```

Unnamed Arguments

- Add `...` as an argument

```
standardize <- function(vector, center = TRUE, scale = TRUE, ...) {
    mean <- mean(vector, ...)
    stdev <- sd(vector, ...)
    if (center) {
        vector <- vector - mean
    }
    if (scale) {
        vector <- vector / stdev
    }
    return(list(result = vector, mean = mean, sd = stdev))
}
```

Apply Our Function to Data

- `airquality` has a column called `Ozone` with missing values

```
airquality$Ozone
```

```
##   [1]  41  36  12  18  NA  28  23  19   8  NA   7  16  11  14  18  14  34   6
##  [19]  30  11   1  11   4  32  NA  NA  NA  23  45 115  37  NA  NA  NA  NA  NA
##  [37]  NA  29  NA  71  39  NA  NA  23  NA  NA  21  37  20  12  13  NA  NA  NA
##  [55]  NA  NA  NA  NA  NA  NA  NA 135  49  32  NA  64  40  77  97  97  85  NA
##  [73]  10  27  NA   7  48  35  61  79  63  16  NA  NA  80 108  20  52  82  50
##  [91]  64  59  39   9  16  78  35  66 122  89 110  NA  NA  44  28  65  NA  22
## [109]  59  23  31  44  21   9  NA  45 168  73  NA  76 118  84  85  96  78  73
## [127]  91  47  32  20  23  21  24  44  21  28   9  13  46  18  13  24  16  13
## [145]  23  36   7  14  30  NA  14  18  20
```

Apply Our Function to Data

- `airquality` has a column called `Ozone` with missing values

```
standard_Ozone <- standardize(airquality$Ozone, na.rm = TRUE)
standard_Ozone$mean
```

```
## [1] 42.12931
```

```
standard_Ozone$sd
```

```
## [1] 32.98788
```

Dealing with . . .

- Note: You can get at the unnamed arguments with `list(...)`

```
f <- function(x, ...){
    unnamed <- list(...)
    modifyX <- x^2
    return(list(newX = modifyX, elipses = unnamed))
}
f(x = 10, a = 1, b = list(char = "hey there", num = 1:3))
```

```
## $newX
## [1] 100
##
## $elipses
## $elipses$a
## [1] 1
##
## $elipses$b
## $elipses$b$char
## [1] "hey there"
##
## $elipses$b$num
## [1] 1 2 3
```

Dealing with . . .

- Alternatively, just grab the names

```
f <- function(x, ...){
    unnamed <- names(list(...))
    modifyX <- x^2
    return(list(newX = modifyX, elipses_names = unnamed))
}
f(x = 10, a = 1, b = list(char = "hey there", num = 1:3))
```

```
## $newX
## [1] 100
##
## $elipses_names
## [1] "a" "b"
```

`tidyverse` Style Functions

(This section is distilled from Modern R with `tidyverse`)

- We've seen the usefulness of functions such as `filter()` and `select()`
- We may want to write functions in a similar manner so they work well with the `tidyverse`
- Specifically, how can we write functions that take columns of data as arguments in the `tidyverse` framework?

Motivation

- Function to find group means

```
iris |>
  group_by(Species) |>
  summarize(across(where(is.numeric),
                   list("mean" = mean),
                   .names = "{.fn}_{.col}"))
```

```
## # A tibble: 3 x 5
##   Species  mean_Sepal.Length mean_Sepal.Width mean_Petal.Length mean_Petal.Width
##   <fct>                <dbl>            <dbl>             <dbl>            <dbl>
## 1 setosa                5.01             3.43              1.46            0.246
## 2 versico~              5.94             2.77              4.26            1.33
## 3 virgini~              6.59             2.97              5.55            2.03
```

Motivation

- Function to find group means

```
find_group_mean <- function(.df, group){
  .df |>
    group_by(group) |>
    summarize(across(where(is.numeric),
                     list("mean" = mean),
                     .names = "{.fn}_{.col}"))
}
find_group_mean(iris, Species)
```

```
## Error in `group_by()`:
## ! Must group by variables found in `.data`.
## x Column `group` is not found.
```

Motivation

- Function to find group means

```
find_group_mean <- function(.df, group){
  .df |>
    group_by(group) |>
    summarize(across(where(is.numeric),
                     list("mean" = mean),
                     .names = "{.fn}_{.col}"))
}
find_group_mean(iris, "Species")
```

```
## Error in `group_by()`:
## ! Must group by variables found in `.data`.
## x Column `group` is not found.
```

Selecting Columns in `tidy` Style Functions

- Two approaches:
  - `enquo()` with `!!()` (injection operator)
  - `{{}}`

Selecting Columns in `tidy` Style Functions

- Two approaches:
  - `enquo()` with `!!()` (injection operator)
  - `{{}}`

```
find_group_mean <- function(.df, group){
  group_name <- enquo(group)
  .df |>
    group_by(!!group_name) |>
    summarize(across(where(is.numeric),
                     list("mean" = mean),
                     .names = "{.fn}_{.col}"))
}
find_group_mean(iris, Species)
```

```
## # A tibble: 3 x 5
##   Species  mean_Sepal.Length mean_Sepal.Width mean_Petal.Length mean_Petal.Width
##   <fct>                <dbl>            <dbl>             <dbl>            <dbl>
## 1 setosa                5.01             3.43              1.46            0.246
## 2 versico~              5.94             2.77              4.26            1.33
## 3 virgini~              6.59             2.97              5.55            2.03
```

Selecting Columns in `tidy` Style Functions

- Two approaches:
  - `enquo()` with `!!()` (injection operator)
  - `{{}}`

```
find_group_mean <- function(.df, group){
  .df |>
    group_by({{group}}) |>
    summarize(across(where(is.numeric),
                     list("mean" = mean),
                     .names = "{.fn}_{.col}"))
}
find_group_mean(iris, Species)
```

```
## # A tibble: 3 x 5
##   Species  mean_Sepal.Length mean_Sepal.Width mean_Petal.Length mean_Petal.Width
##   <fct>                <dbl>            <dbl>             <dbl>            <dbl>
## 1 setosa                5.01             3.43              1.46            0.246
## 2 versico~              5.94             2.77              4.26            1.33
## 3 virgini~              6.59             2.97              5.55            2.03
```

Combining with . . .

- We can allow for multiple columns with . . .
- Must use `quos()` and `!!!()` instead

```
find_group_mean <- function(.df, ...){
  group_vars <- quos(...)
  .df |>
    group_by(!!!group_vars) |>
    summarize(across(where(is.numeric),
                     list("mean" = mean),
                     .names = "{.fn}_{.col}"))
}
find_group_mean(CO2, Type, Treatment)
```

```
## # A tibble: 4 x 4
## # Groups:   Type [2]
##   Type        Treatment  mean_conc mean_uptake
##   <fct>       <fct>          <dbl>       <dbl>
## 1 Quebec      nonchilled       435        35.3
## 2 Quebec      chilled          435        31.8
## 3 Mississippi nonchilled       435        26.0
## 4 Mississippi chilled          435        15.8
```

`as_label()` for `tidyverse` Style Functions

- We may want to name a variable using a column passed
- `as_label()` can be used!
- Must use "Walrus" operator, `:=`

```
find_group_mean <- function(.df, group, column){
  group_name <- enquo(group)
  column_name <- enquo(column)
  column_label <- paste0("mean_", as_label(column_name))
  .df |>
    group_by(!!group_name) |>
    summarize(!!(column_label) := mean(!!column_name))
}
find_group_mean(iris, Species, Sepal.Length)
```

```
## # A tibble: 3 x 2
##   Species    mean_Sepal.Length
##   <fct>                  <dbl>
## 1 setosa                  5.01
## 2 versicolor              5.94
## 3 virginica               6.59
```

Pipeable functions

- Piping is great - we may want to make sure our functions are pipeable!
- Two types of pipeable functions:

    1. **transformations**
    2. **side-effects**

Pipeable functions

- Piping is great - we may want to make sure our functions are pipeable!
- Two types of pipeable functions:

  1. **transformations**
  2. **side-effects**

- transformations naturally return the modified argument (df)
- side-effects don't
  - Solution: Silently return the DF with `invisible()`

Pipeable functions

- Example: Side-effect function to print info

```
print_num_obs <- function(.df) {
    cat("The number of observations in the data set is ",
        nrow(.df),
        "\n",
        sep = "")
}
iris |>
  print_num_obs() |>
  summarize(mean = mean(Sepal.Length))
```

```
## The number of observations in the data set is 150
## Error in UseMethod("summarise"): no applicable method for 'summarise' applied to an object of class "NULL"
```

Pipeable functions

- Example: Side-effect function to print info

```
print_num_obs <- function(.df) {
    cat("The number of observations in the data set is ",
        nrow(.df),
        "\n",
        sep = "")
  invisible(.df)
}
iris |>
  print_num_obs() |>
  summarize(mean = mean(Sepal.Length))
```

```
## The number of observations in the data set is 150
##       mean
## 1 5.843333
```

Lazy Evaluation

- R evaluates arguments only when needed!
- Consider the silly function below:

```
run <- function(x){
  3
}
run(stop("stop now!"))
```

```
## [1] 3
```

Lazy Evaluation

- R evaluates arguments only when needed!
    - Force evaluation by writing the argument or `force(arg)`

```
run <- function(x){
  force(x) #or just x, this just makes it explicit it wasn't a typo!
  3
}
run(stop("stop now!"))
```

```
## Error in force(x): stop now!
```

Lazy Evaluation On Comparisons

- This is true for compound `if` statements as well!

```
x <- NULL
x > 0
```

```
## logical(0)
```

```
if(x > 0){
  print("hey")
}
```

```
## Error in if (x > 0) {: argument is of length zero
```

```
!is.null(x)
```

```
## [1] FALSE
```

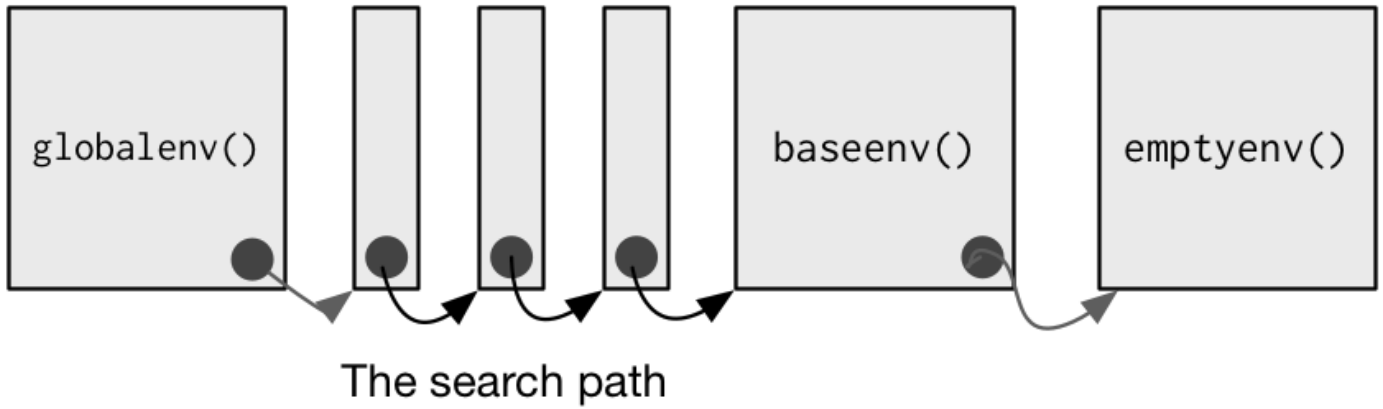
```
if (!is.null(x) && x > 0) {
  print("hey")
}
```

Environments and Lexical Scoping

- R objects live in an environment
- You can think of it as a "bag of names" that point to things in memory
- Like a list but with no ordering (and other things)

Environments and Lexical Scoping

- Environments have 'parents' and 'children'
  - Global environment is where our created function objects live
  - Search path has all packages loaded in (most recent package is the parent of the global environment)
  - Base environment is the child of the ultimate ancestor, the empty environment



Environments and Lexical Scoping

- We can see the 'search' path using `search()`

```
##  [1] ".GlobalEnv"        "package:knitr"     "package:forcats"
##  [4] "package:stringr"   "package:dplyr"     "package:purrr"
##  [7] "package:readr"     "package:tidyr"     "package:tibble"
## [10] "package:ggplot2"   "package:tidyverse" "package:stats"
## [13] "package:graphics"  "package:grDevices" "package:utils"
## [16] "package:datasets"  "package:methods"   "Autoloads"
## [19] "package:base"
```

Environments and Lexical Scoping

- Don't need to fully understand environments but some things are important

```
library(pryr) #install if needed
x <- "hey"
where("x")
```

```
## <environment: R_GlobalEnv>
```

```
where("mean")
```

```
## <environment: base>
```

Environments and Lexical Scoping

- When you call a function, it creates temporary function environments
- This is why variables in functions don't overwrite things!

```
f <- function(x){
  mean <- paste0(x, " is a value")
  mean
}
f(1:3)
```

```
## [1] "1 is a value" "2 is a value" "3 is a value"
```

```
mean
```

```
## function (x, ...)
## UseMethod("mean")
## <bytecode: 0x00000000160cb700>
## <environment: namespace:base>
```

Environments and Lexical Scoping

- When you call a function, it creates temporary function environments
- This is why variables in functions don't exist outside the function

```
g <- function(x) {
  if (!exists("a", inherits = FALSE)) {
    message("Defining a")
    a <- 1
  } else {
    a <- a + 1
  }
  a
}
g(10)
```

```
## [1] 1
```

```
g(10)
```

```
## [1] 1
```

Environments and Lexical Scoping

- When you call a function, it creates temporary function environments
- This is why variables can have the same name in a function and in your global environment

```
y <- 10
f <- function(x){
  y <- 1
  x + y
}
f(15)
```

```
## [1] 16
```

Environments and Lexical Scoping

- **Important**: If R doesn't find an object in the current environment, it will search up the path

```
y <- 1
f <- function(x){
    x + y
}
f(10)
```

```
## [1] 11
```

Recap!

- `...` for unnamed arguments
- Writing `tidyverse` style functions
- Lazy evaluation
- Environments and lexical scoping