

Stat405

Functions & for loops

Hadley Wickham

1. Changes to homework
2. Function structure and calling conventions
3. Practice writing functions
4. For loops

Homework changes

Figuring out the problem isn't as easy as I'd thought, so we'll cover it in class today.

Due next Wednesday, with some TBA function drills.

Pay attention to the style guide!

Basic structure

- Name
- Input arguments
 - Names/positions
 - Defaults
- Body
- Output (final result)

```

calculate_prize <- function(windows) {
  name <- c("DD" = 800, "BB" = 40,
            25, "B" = 10, "arguments" = 0)

  same <- length(unique(windows)) == 1
  allbars <- all(windows %in% c("B", "BB", "BBB"))

  if (same) {
    prize <- payoffs[windows[1]]
  } else if (allbars) {
    prize <- 5
  } else {
    prize <- sum(windows == "C")
    prize <- sum(windows == "DD")

    prize <- c(0, 2, 5)[cherries + 1] *
      c(1, 2, 4)[diamonds + 1]
  }
  prize
}

```

always indent
inside {}!

last value in function is result

```
mean <- function(x) {  
  sum(x) / length(x)  
}
```

```
mean(1:10)
```

default value

```
mean <- function(x, na.rm = TRUE) {  
  if (na.rm) x <- x[!is.na(x)]  
  sum(x) / length(x)  
}
```

```
mean(c(NA, 1:9))
```

```
# Function: mean  
str(mean)  
mean
```

```
args(mean)  
formals(mean)  
body(mean)
```

Function arguments

Every function argument has a name and a position. When calling, matches **exact name**, then **partial name**, then **position**.

Rule of thumb: for common functions, position based matching is ok for required arguments. Otherwise use names (abbreviated as long as clear).

```
mean(c(NA, 1:9), na.rm = T)
```

```
# Confusing in this case, but often saves typing  
mean(c(NA, 1:9), na = T)
```

```
# Generally don't need to name all arguments  
mean(x = c(NA, 1:9), na.rm = T)
```

```
# Unusual orders best avoided, but  
# illustrate the principle  
mean(na.rm = T, c(NA, 1:9))  
mean(na = T, c(NA, 1:9))
```

```
# Overkill
```

```
qplot(x = price, y = carat, data = diamonds)
```

```
# Don't need to supply defaults
```

```
mean(c(NA, 1:9), na.rm = F)
```

```
# Need to remember too much about mean
```

```
mean(c(NA, 1:9), , T)
```

```
# Don't abbreviate too much!
```

```
mean(c(NA, 1:9), n = T)
```

```
f <- function(a = 1, abcd = 1, abdd = 1) {  
  print(a)  
  print(abcd)  
  print(abdd)  
}
```

```
f(a = 5)
```

```
f(ab = 5)
```

```
f(abc = 5)
```

Your turn

Write a function to calculate the variance of a vector. Write a function to calculate the covariance of two vectors.

Make sure each has a `na.rm` argument.

Write functions in your text editor, then copy into R.

Strategy

Always want to start simple: start with test values and get the body of the function working first.

Check each step as you go.

Don't try and do too much at once!

Create the function once everything works.

```
x <- 1:10
```

```
sum((x - mean(x)) ^ 2)
```

```
var <- function(x) sum((x - mean(x)) ^ 2)
```

```
x <- c(1:10, NA)
```

```
var(x)
```

```
na.rm <- TRUE
```

```
if (na.rm) {
```

```
  x <- x[!is.na(x)]
```

```
}
```

```
var <- function(x, na.rm = F) {
```

```
  if (na.rm) {
```

```
    x <- x[!is.na(x)]
```

```
  }
```

```
  sum((x - mean(x)) ^ 2)
```

```
}
```

Testing

Always a good idea to test your code.

We have a prebuilt set of test cases: the prize column in slots.csv

So **for** each row in slots.csv, we need to calculate the prize and compare it to the actual. (Hopefully they will be same!)

For loops

```
for(value in 1:10) {  
  print(value)  
}
```

```
cuts <- levels(diamonds$cut)  
for(cut in cuts) {  
  selected <- diamonds$price[diamonds$cut == cut]  
  print(mean(selected))  
}  
# Have to do something with output!
```

```
# Common pattern: create object for output,  
# then fill with results  
  
cuts <- levels(diamonds$cut)  
means <- vector("numeric", length(cuts))  
  
for(i in seq_along(cuts)) { Why use i and not cut?  
  sub <- diamonds[diamonds$cut == cuts[i], ]  
  means[i] <- mean(sub$price)  
}  
  
# We will learn more sophisticated ways to do this  
# later on, but this is the most explicit
```

```
seq_len(5)
```

```
seq_len(10)
```

```
n <- 10
```

```
seq_len(10)
```

```
seq_along(1:10)
```

```
seq_along(1:10 * 2)
```

Your turn

For each diamond colour, calculate the median price **and** carat size

```
colours <- levels(diamonds$color)
n <- length(colours)
mprice <- rep(NA, n)
mcarat <- rep(NA, n)

for(i in seq_len(n)) {
  set <- diamonds[diamonds$color == colours[i], ]
  mprice[i] <- median(set$price)
  mcarat[i] <- median(set$carat)
}

results <- data.frame(colours, mprice, mcarat)
```

Back to slots...

For each row, calculate the prize and save it, then compare calculated prize to actual prize

Question: given a row, how can we extract the slots in the right form for the function?

```
slots <- read.csv("slots.csv")
```

```
i <- 334
```

```
slots[i, ]
```

```
slots[i, 1:3]
```

```
str(slots[i, 1:3])
```

```
as.character(slots[i, 1:3])
```

```
c(as.character(slots[i, 1]), as.character(slots[i, 2]),  
as.character(slots[i, 3]))
```

```
slots <- read.csv("slots.csv", stringsAsFactors = F)
```

```
str(slots[i, 1:3])
```

```
as.character(slots[i, 1:3])
```

```
calculate_prize(as.character(slots[i, 1:3]))
```

```
# Create space to put the results
slots$check <- NA

# For each row, calculate the prize
for(i in seq_len(nrow(slots))) {
  w <- as.character(slots[i, 1:3])
  slots$check[i] <- calculate_prize(w)
}

# Check with known answers
subset(slots, prize != prize_c)
# Uh oh!
```

What is the problem? Think
about the most generalise case

DD	DD	DD	800
7	7	7	80
BBB	BBB	BBB	40
B	B	B	10
C	C	C	10
Any bar	Any bar	Any bar	5
C	C	*	5
C	*	C	5
C	*	*	2
*	C	*	2
*	*	C	2

```

windows <- c("DD", "C", "C")
# How can we calculate the
# payoff?

```

DD doubles any winning combination. Two DD quadruples. **DD is wild**