class: center, middle

Programming in R

Adam Rawles

Recap

–

- Loading data from .csv and .xlsx

–

- Cleaning data
  - `is/as.xxxxxx()` functions

–

- Summary statistics

–

- `mean()`, `median()`, `sd()`...
- `summary()`

–

- Plotting

–

- `plot()`
- `hist()`

Overview

- User-defined functions

–

- For loops

–

- If/else statements

Functions

- Functions are how we perform any action in R

–

- We pass arguments to a function as an input, there is some form of transformation, and we get an output

–

- For example, `read.csv()` takes a .csv file, and turns it into a dataframe

–

- There are thousands of predefined functions in base R, and even more with packages

–

- But often, you'll need a specific function for a specific task…

–

- And that's where user-defined functions come in

User-defined functions - basics

- Functions can have multiple inputs, but only one output

–

- Functions are named*, and are always followed by () when used

–

- Functions names should be unique but memorable/logical

–

- Functions should be as simple and applicable as possible

–

- Functions will return the last evaluated (not assigned) variable, or whatever is included in the `return()` call

–

- Functions can only return *one* thing (but vectors and lists still count as one thing)

–

* There are unnamed functions called "anonymous functions" but that's for a different module

---

User-defined functions - structure

- Functions are created with the following structure:

–

```r
some_function <- function(some_input, ...){
  operation
  return(return_value)
}
```

–

- We name the function, define what inputs we want, and then what we want to do with those inputs

---

User-defined functions - example

–

```r
sum_custom <- function(x, y){
  new_value <- x + y
  return(new_value)
}

sum_custom(1,2)
```

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

User-defined functions - example

This same function can also be written without an explicit `return()` call:

–

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

sum_custom(1,2)
```

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

User-defined functions - example

–

- You can specify default values for any of your input parameters, making that argument optional:

–

```r
combine_custom <- function(string1, string2, delimiter = " ") {
  new_string <- paste0(string1, delimiter, string2)
  return(new_string)
}

combine_custom(string1 = "hello", string2 = "world")
```

```
## [1] "hello world"
```

User-defined functions - example

- But you can override the default by providing a value to that parameter when using the function

–

```r
combine_custom(string1 = "hello",
               string2 = "world",
               delimiter = "!")
```

```
## [1] "hello!world"
```

User-defined functions - exercise

- Option 1 (easy)
  - Write a function that takes 2 numbers, multiplies them together and divides the result by 2

- Option 2 (intermediate) - Write a function that takes 2 vectors, and multiplies the largest value of vector 1 by the smallest value in vector 2

- Option 3 (advanced)
  - Write a function that takes 2 strings, and returns the 1st and 2nd values of each one (hint available)

–

- Option 4 (theoretical)
  - Think of how you would write a function that can sum an indefinite number of numbers

---

User-defined functions - answers

- Option 1
    - Write a function that takes 2 numbers, multiplies them together and divides the result by 2

```
option1_function <- function(x,y){
  new_val <- (x * y)/2
  return(new_val)
}

option1_function(4,4)
```

```
## [1] 8
```

User-defined functions - answers

- Option 2
    - Write a function that takes 2 vectors, and multiplies that largest value of vector 1 by the smallest value in vector 2

```
option2_function <- function(v1, v2){
  new_val <- max(v1) * min(v2)
  return(new_val)
}

option2_function(v1 = c(1,2,3,4), v2 = c(1,2,3,4,5))
```

```
## [1] 4
```

User-defined functions - answers

- Option 3
    - Write a function that takes 2 strings, and returns the 1st and 2nd values of each one

```
option3_function <- function(string1, string2){
  ret1 <- substr(string1,1,2)
  ret2 <- substr(string2, 1,2)
  ret <- c(ret1, ret2)
  return(ret)
}

option3_function("hello", "world")
```

```
## [1] "he" "wo"
```

User-defined functions

- In some cases, you may want to provide a variable number of inputs to a function (like if you want to add an indefinite number of values together)

–

- For example, if you have a function that combines strings, you may want to accept any number of strings to combine

–

- To do this, we use the ellipsis (...) argument when defining our function

```r
some_function <- function(...){
  arguments <- list(...)
  return(arguments)
}
```

User-defined functions

```r
some_function("hello", "world")
```

```
## [[1]]
## [1] "hello"
##
## [[2]]
## [1] "world"
```

```r
some_function("hello", "world", "again")
```

```
## [[1]]
## [1] "hello"
##
## [[2]]
## [1] "world"
##
## [[3]]
## [1] "again"
```

User-defined functions - environment

–

- Functions have a local environment, meaning that anything calculated in the function is not accessible outside the function (except the value that is returned)

–

```
some_function <- function(x, y){
  m <- x * y
  s <- x + y
  return(m)
}

some_function(1, 2)
```

```
## [1] 2
```

```
print(s)
```

```
## Error in print(s): object 's' not found
```

–

-

  This code errors, because the `s` variable isn't accessible outside of the function

User-defined functions - returning

- As previously mentioned, a function will return the last evaluated object, or whatever is returned via the `return()` function.

–

- What's really important to remember however, is that a function will return a *copy* of the return value, not the return object

–

```
some_function <- function(x) {
  return(x + 1)
}

x <- 1
some_function(x)
```

```
## [1] 2
```

```r
x
```

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

User-defined functions - returning

- To change the original object, we need to reassign the result of the function back to the object…

–

```r
x <- 1
x <- some_function(x)
x
```

```
## [1] 2
```

For loops

–

- Sometimes, we may want to repeat the same action more than once

–

- For example, we might want a function to add 4 to every item in a vector, or get the mean for every column in a dataframe

–

- You could copy and paste the code required each time, or you could use a for loop

---

For loops

- With a for loop, you can iterate over ever item in a list or vector and perform an action

–

- While loops (which iterate until a condition is met) also exist, but we're going to focus on for loops

–

- For loops follow a basic structure:

```
for (identifier in list or vector){
  what we want to do with each item
}
```

–

- The identifier becomes the variable name for accessing the current value in the body of the loop

–

- On each iteration, the identifier variable will take on a new value

---

For loops - structure

- You can also perform a for loop a defined number of times rather than iterating through a list/vector:

```
for (identifier in seq_along(1:some_number)){
  what we want to do with each item
}
```

- In this case, the value of our identifier variable will change to the next number in our set of numbers

---

For loops - example

- Say we want to loop through a vector and print each value…

–

```
vector1 <- c(1,2,3,4,5,6,7,8)
for (i in vector1){
  print(i)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
```

For loops - excercise

- Option 1 (easy)
  - Write a for loop that divides each number in a vector of numbers by 2

–

- Option 2 (intermediate)
  -

Write a for loop that produces a running average from a vector of numbers

- Option 3 (advanced)
  - Write a for loop that adds each value from one vector to the value at the next index in a second vector

---

For loops - answers

- Option 1
  - Write a for loop that divides each number in a vector of numbers by 2

```
vector1 <- c(2,4,6,8,10)
for (i in vector1){
  print(i/2)
}
```

```
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
```

For loops - answers

- Option 2
  - Write a for loop that produces a running average from a vector of numbers

```
vector1 <- c(10,20,30,40,70,100)
total <- 0
counter <- 0
for (i in vector1){
  counter <- counter + 1
  total <- total + i
 print(total/counter)
}
```

```
## [1] 10
## [1] 15
## [1] 20
## [1] 25
## [1] 34
## [1] 45
```

For loops - answers

- Option 3
  - Write a for loop that adds each value from one vector to the value at the next index in a second vector

```r
vector1 <- c(1,5,10,15)
vector2 <- c(2,6,10,14,18)

for (i in seq_along(vector)){
  print(vector1[i] + vector2[i+1])
}
```

```
## [1] 7
```

If else statements

- Sometimes, you'll only want to perform an action if a certain criteria is met

–

- For example, you may only want to add 4 to a number if it's greater than 10

–

- To perform a certain action based on mutliple criteria, you use an if else statement

---

If else statements - structure

- There are 3 main 'types' of if else statements

–

- Simple if statements
  - If the criteria is fulfilled, perform the action, otherwise do nothing:

–

```
if (criteria){
  do something
}
```

–

- If else statements
  - If the criteria is fulfilled, perform the action, otherwise do something else:

–

```
if (criteria){
  do something
} else {
  do something else
}
```

If else statements - structure

- If and if else statements
  - If the criteria is fulfilled perform the action, otherwise if a different criteria is fulfilled do something else, otherwise do nothing:

–

```
if (criteria){
  do something
} else if (other criteria) {
  do something else
}
```

If else statements - criteria

- You can include multiple criteria in one if or else statement…

–

```
if (criteria 1 | criteria 2){

}

if (criteria 1 & criteria 2){

}
```

If else statements - example

–

```
vector1 <- c(1,2,3,4,5)

for (i in vector1){
  if (i == 1) {
    print("The value is 1")
  } else if (i == 2) {
    print("The value is 2")
  } else if (i == 3 | i == 4){
    print("The value is 3 or 4")
  } else {
    print("The value is not 1, 2, 3 or 4")
  }
}
```

```
## [1] "The value is 1"
## [1] "The value is 2"
## [1] "The value is 3 or 4"
## [1] "The value is 3 or 4"
## [1] "The value is not 1, 2, 3 or 4"
```

Final - exercise

- Option 1 (easy)
  - Write a function that loops through a vector of numbers and returns only the even ones
  - Hint: you can use x %% y to check if a number is even

–

- Option 2 (intermediate)
  - Write a function that loops through a vector of numbers and returns the values that are smaller than the value at the same index in a second vector
  - You can assume that the two vectors will always be the same length

–

- Option 3 (advanced)
  - Write a function that loops through a vector of numbers and square it if it is a multiple of 4, otherwise replace the value with the previous value in the vector or 0 if the value is first in the vector and return
  - Hint: use x / y %% 2 to check for multiples

Final - answers

- Option 1
  - Write a function that loops through a vector of numbers and returns only the even ones

```r
option1_function <- function(v) {
  return_vector <- c()
  for (i in v){
    if (i %% 2 == 0){
      return_vector <- append(return_vector, i)
    }
  }
}

option1_function(v = c(1,2,3,4,5,6,7,8))
```

Final - answers

- Option 2
  - Write a function that loops through a vector of numbers and returns the values that are smaller than the value at the same index in a second vector

```r
option2_function <- function(v1, v2) {
  return_vector <- c()
  for (i in seq_along(v1)){
    if (v1[i] < v2[i]){
      return_vector <- append(return_vector, v1[i])
      }
  }
}

option2_function(v1 = c(1,2,3,4,5,6,7,8), v2 = c(2,1,4,5,2,1,1,1))
```

Final - answers

- Option 3
  - Write a function that loops through a vector of numbers and squares it if it is a multiple of 4, otherwise replace the value with the previous value in the vector or 0 if the value is first in the vector and return

---

Final - answers

```r
option3_function <- function(v) {
  for (i in seq_along(v)){
    if ((v[i] / 4) %% 2 == 0){
      v[i] <- v[i]^2
    } else {
        if (i == 1){
         v[i] = 0
        } else {
          v[i] = v[i-1]
        }
    }
  }
  return(v)
}

option3_function(v = c(1,2,3,4,5,6,7,8))
```

```
## [1]  0  0  0  0  0  0  0 64
```

Conclusion

–

- User-defined functions
  - We use functions to perform repeatable and generalizable tasks

–

- For loops
  - We use for loops to iterate over vectors/lists, or to perform an action a certain number of times

–

- If else statements
  - With if, else if, and else statements, we can perform actions only when a certain criteria is met

---

Future modules (optional)

- Statistical analysis
- Simulations
- Improving efficiency