

Functional Programming in R

Martha Cooper, `tRopical_R`

James Cook University::ALTHM

2019/06/6 (updated: 2020-09-06)

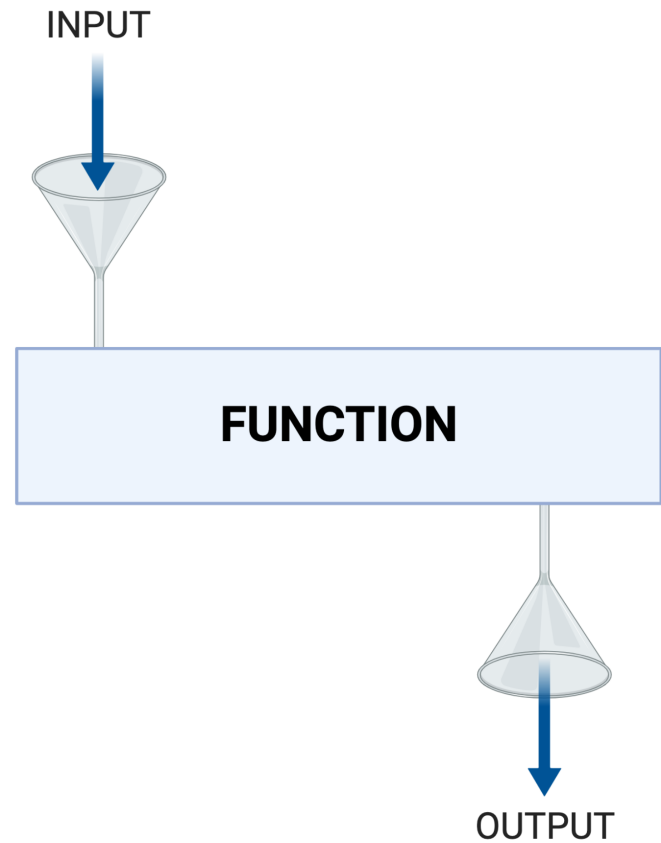
Outline

1. Functions
2. Iteration (Loops)
3. Conditional Statements

Functions

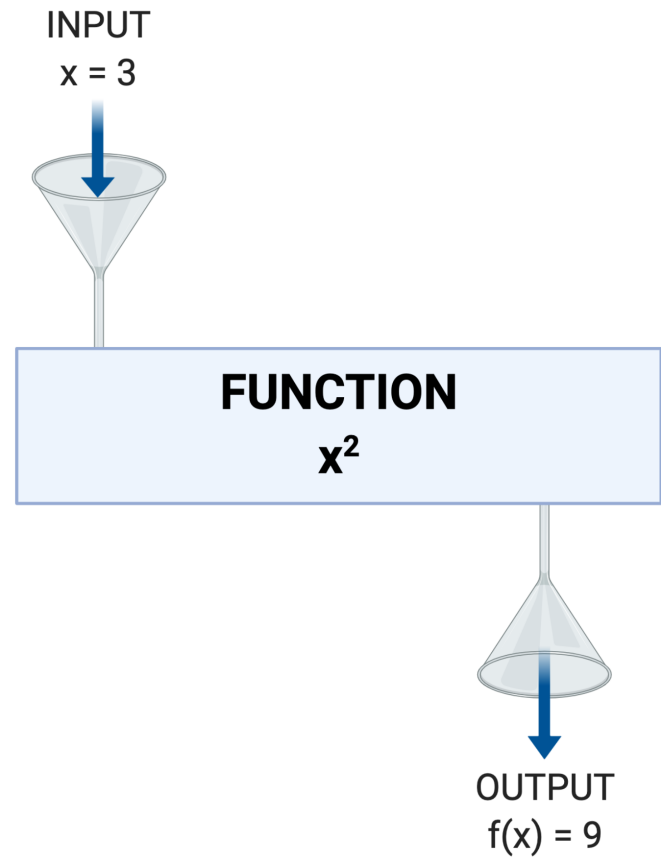
What is a function?

- A function is a reusable piece of code that does a specific task



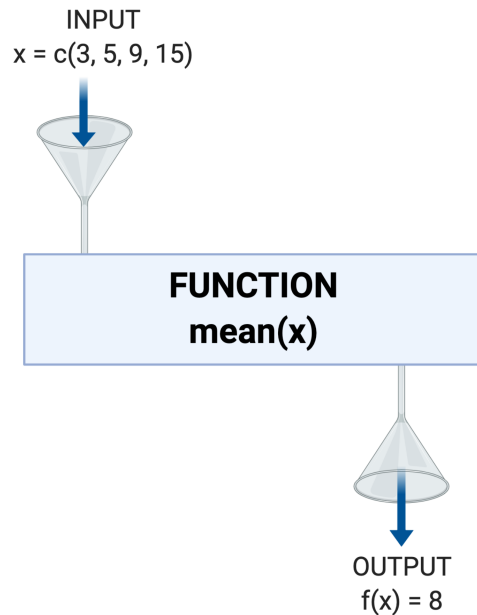
What is a function?

- A function is a reusable piece of code that does a specific task



R has built in functions

- Schematic



- R code

```
#input  
x <- c(3, 5, 9, 15)
```

```
#function  
mean(x)
```

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

Functions take arguments

- Function inputs are called arguments
- **mean()** has 3 arguments
- **?mean** to view arguments
- Arguments can be specified by *order* or *name*

```
x # a numeric vector  
  
trim # fractions of outliers  
      # you would like to  
      # trim from each end  
  
na.rm # option to remove  
       # missing values
```

Packages also contain functions

```
library(ggplot2)
ggplot(data = mtcars, aes(x = as.factor(cyl), y = mpg))+
  geom_boxplot()+
  xlab("cyl")
```


Function Syntax

A function is defined as follows:

```
add_1 <- function(x){  
  x + 1  
}
```

We can break this down into:

- The **signature** (the user interface)

```
add_1 <- function(x)
```

- The **body** (the code the function executes)

```
{  
  x + 1  
}
```

3 Reasons to write functions

1. Code becomes easier to read & understand

Functions have evocative names e.g. `mean()`, `sum()`, `sd()`

2. Code becomes easier to change

`add_1 <- function(x){x + 1}` can easily become `add_2 <- function(x){x + 2}`

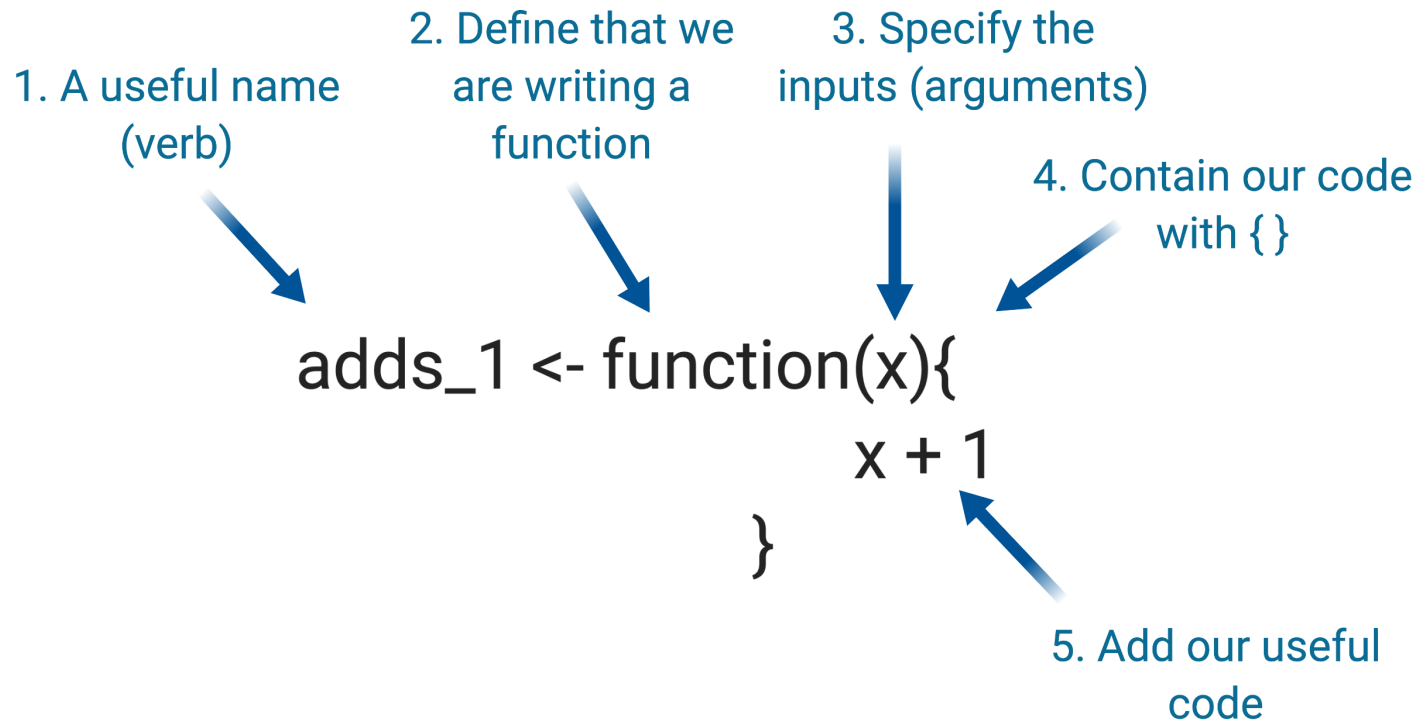
3. Less mistakes

No copy and pasting chunks of code

"You should consider writing a function whenever you've copied and pasted a block of code more than twice"

Hadley Wickham

Writing your own function (Step by Step)



Embedding Functions

This code:

```
exp_1 <- read.csv("exp_1_data.csv") #import  
exp_1_filtered <- filter(exp1, value < 10) #filter
```

Can be wrapped into a function:

```
#write a function to import and filter data  
import_and_filter <- function(path_to_file){  
  
  x <- read.csv(path_to_file) #import  
  x <- filter(x, value < 10) #filter  
  return(x)  
}
```

Now we can repeat for lots of experiments:

```
#Use function  
exp_1 <- import_and_filter("exp_1_data.csv")  
exp_2 <- import_and_filter("exp_2_data.csv")  
exp_3 <- import_and_filter("exp_3_data.csv")
```

Exercise 1

- Write your own function that multiplies the input by 10
- Answer

```
times10 <- function(x){  
  10*x  
}
```

Documentation

- Documentation is essential, even if the only person reading your code will be you in the future
- Check help for any function for the best examples (?mean)

```
#multiplies the input by ten  
#input must be numeric  
times10 <- function(x){  
  10*x  
}
```

Exercise 2

- Embed the following functions in your own function

```
my_data <- read.csv("my_csv_file.csv")  
my_processed_data <- mutate(my_data, total = sum)
```

- Answer

```
input_and_process <- function(my_csv_file){  
  my_data <- read.csv(my_csv_file)  
  my_processed_data <- mutate(my_data, total = sum(my_column))  
  return(my_processed_data)  
}
```


Iteration (Loops)

Iteration (Loops)

+ Like functions, iteration (loops) is (are) another tool to *reduce duplication* in your code, resulting in code that is *easier to read* with *reducing mistakes*.

For Loops

Say we want to calculate the mean of every column in a dataframe (we will use the mtcars dataframe as an example:)

```
head(mtcars, 3)
```

```
##           mpg cyl  disp  hp  drat    wt  qsec vs am gear carb
## Mazda RX4      21.0   6  160 110  3.90  2.620 16.46  0  1    4    4
## Mazda RX4 Wag  21.0   6  160 110  3.90  2.875 17.02  0  1    4    4
## Datsun 710     22.8   4  108  93  3.85  2.320 18.61  1  1    4    1
```

We could calculate each mean one by one, like this:

```
mean(mtcars$mpg) #20.09062
mean(mtcars$cyl) #6.1875
mean(mtcars$disp) #230.7219
# ...etc ...
```

For Loops

A better (think *less errors* and *easier to read*) approach is to use a *For Loop*

```
mtcars_means <- vector(length = ncol(mtcars)) # 1. Define Output

for (i in seq_along(mtcars)) {                # 2. Define Sequence
  mtcars_means[[i]] <- mean(mtcars[[i]])      # 3. Define Body
}

mtcars_means
```

```
## [1] 20.090625 6.187500 230.721875 146.687500 3.596563 3.217250 17.
## [10] 3.687500 2.812500
```

Loops have 3 parts:

- Before you start the loop, you must define a container for the output. A good way to do this is to create an empty vector the same length as our dataframe we want to iterate through:

```
mtcars_means <- vector(length = ncol(mtcars)) # 1. Define Output
```

- We then define what to iterate/loop through. Each pass of the loop will assign i to a different value from `seq_along(mtcars)` - in this case each value of i is a column of `mtcars`:

```
for (i in seq_along(mtcars)) # 2. Define sequence
```

- The body is the code that we want to execute for every value of i . It is run repeatedly, in this case for each column of `mtcars`:

```
mtcars_means[[i]] <- mean(mtcars[[i]])
```

Exercise 3

- Write a for loop to calculate the median of every column in the USArrests dataset
- Answer

```
USArrests_medians <- vector(length = ncol(USArrests)) # 1. Define Output  
  
for (i in seq_along(USArrests)) {                      # 2. Define Sequence  
  USArrests_medians[[i]] <- median(USArrests[[i]])      # 3. Define Body  
}
```

```
USArrests_medians
```

```
## [1] 7.25 159.00 66.00 20.10
```

Conditional Statements

If Statements

If Statements are conditional statements in R. In an If Statement, we define code to be executed **if** certain conditions are met

```
x <- 5

if(x > 0){                                # 1. Define Condition
  print("Positive Number")                # 2. Define Body
}
```

```
## [1] "Positive Number"
```


If..Else Statements

But what if $x < 0$

```
x <- -4

if(x > 0){                                # 1. Define Condition
  print("Positive Number")                # 2. Define Body
}
```

- Nothing happens... unless we add else

```
x <- -4

if(x > 0){                                # 1. Define Condition
  print("Positive Number")                # 2. Define Body
} else {
  print("Negative Number")
}
```

```
## [1] "Negative Number"
```

Side Note: ifelse() is a fast alternative

- For simple cases, we can use the base R function ifelse() instead

```
x <- 5  
  
my_test <- ifelse(test = x>0,  
                  yes = print("Positive Number"),  
                  no = print("Negative Number"))
```

```
## [1] "Positive Number"
```

```
#This means: if x > 0 ,  
#             then print "Positive Number",  
#             or if not print "Negative Number"
```

```
my_test
```

```
## [1] "Positive Number"
```

Exercise 4

```
x <- 10  
y <- 5
```

- Define an condition statement that prints "x times y is big" if $x*y > 10$
- Answer

```
if(x*y > 10){  
  print("x times y is big")  
}
```

```
## [1] "x times y is big"
```

Exercise 4

- Redefine x and y as follows:

```
x <- 0.8  
y <- 2
```

- Add a useful else... condition to our if else statement from before
- Answer

```
if(x*y > 10){  
  print("x times y is big")  
} else {  
  print("x times y is small")  
}
```

```
## [1] "x times y is small"
```

Exercise 5

```
x <- c(3,-4,6,-9)
```

- Using a For Loop, iterate through the vector x and print whether each value is positive or negative
- Answer

```
my_answers <- vector(mode = "character", length = length(x))  
  
for (i in seq_along(x)){  
  if(x[i] > 0){  
    my_answers[i] = "X is positive"  
  } else {  
    my_answers[i] = "X is negative"  
  }  
}  
  
my_answers
```

```
## [1] "X is positive" "X is negative" "X is positive" "X is negative"
```

Acknowledgements:

- Some material used in this presentation has been modified from <https://r4ds.had.co.nz/>
- Slides created via the R package **xaringan**
- Theme created with the R package **xaringanthemer**
- Help has come from **remark.js**, **knitr**, and **R Markdown**.