Practical R: Functions and Loops

Abhijit Dasgupta

BIOF 339

Functions

Why do we need functions?

When you are typing instructions to the computer, you might find yourself repeating the same instructions over and over. So you end up copying and pasting code for each repitition.

- Can make a mistake copying and pasting
- If you need to change the instructions, you need to find every instance of it **manually** and change it, and you're likely to miss one

The rule of thumb is, if you're copying the same code more than twice, write a function.

- Write the instructions once
- Change it in only one place, if needed

Defining functions

The basic syntax of a function is

```
<function name> <- function(<input argument(s)>){
  <code for instructions>
  ...
  <more code>
  return(<output object>)
}
```

Defining functions

Let's create our own function to convert feet to inches.

```
ft2in <- function(ft){
  inch <- ft * 12
  return(inch)
}
```

- `ft2in` is the name of the function
- The input argument is named `ft` (make an expressive name)
- Inches are computed by multiplying `ft` by 12 and storing it in `inch`
- The output of the function is the value of the `inch` variable

To run this:

```
ft2in(12) # 12 feet to inches
```

```
[1] 144
```

Defining functions

What if we want more than one input?

```
ft2in <- function(ft, convert_to){
  # ft = input (feet)
  # convert_to = unit to convert to ('in','m','cm')
  if(convert_to == 'in'){
    output <- ft * 12
  }
  if(convert_to == 'm'){
    output <- ft * 0.3048
  }
  if(convert_to == 'cm'){
    output <- ft * 30.48
  }
  return(paste(output, convert_to))
}
```

```
ft2in(12, convert_to='cm')
```

```
[1] "365.76 cm"
```

Quick reminder about conditions

Some comparison operators for filtering

Operator	Meaning
==	Equals
!=	Not equals
> / <	Greater / less than
>= / <=	Greater or equal / Less or equal
!	Not
%in%	In a set

Combining comparisons

Operator	Meaning
&	And
|	Or

dplyr::case_when()

IF ELSE... (but you love it?)

```
df %>% ADD COLUMN 'danger'
  mutate(danger = case_when(type == "kraken" ~ "extreme!",
                            TRUE ~ "high"))
```

IF type is kraken THEN danger is extreme!

OTHERWISE, danger is high.

type	age	danger
kraken	baby	extreme!
dragon	adult	high
cyclops	teen	high
kraken	adult	extreme!
dragon	teen	high

Defining functions

```
ft2in <- function(ft, convert_to){
  # ft = input (feet)
  # convert_to = unit to convert to ('in','m','cm')
  conversion <- case_when(
    convert_to == 'in' ~ 12,
    convert_to =='m' ~ 0.3048,
    convert_to == 'cm' ~ 30.48,
    TRUE ~ 1  # otherwise
  )
  output = ft * conversion
  return(paste(output, convert_to))
}
```

```
ft2in(12, convert_to='cm')
```

```
[1] "365.76 cm"
```

The concept of local vs global variables

```
x <-  10
print(x)
```

```
[1] 10
```

```
f <- function(x){
  x <- 5
  print(x)
}

f(x)
```

```
[1] 5
```

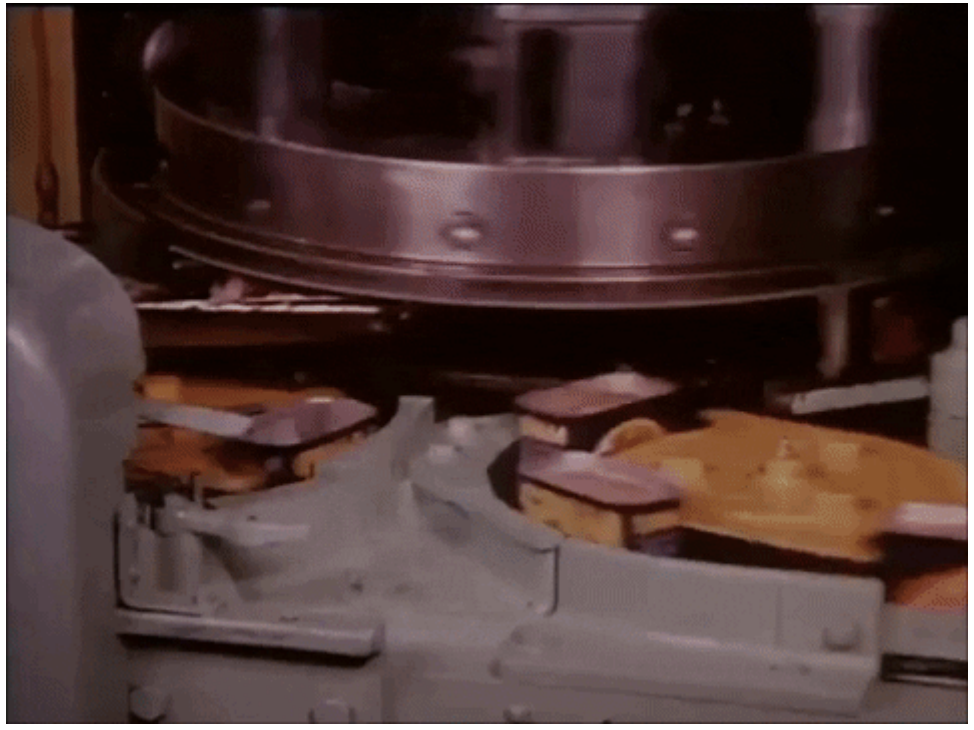
```
print(x)
```

```
[1] 10
```

The x inside the function is local to the function and is independent of the x in the global space that has the value 10..

Loops

for-loops



for-loops

The for-loop is a construct to repeat the same operation over a list of values.

Basic syntax:

```
for(<variable> in <list>){
    <code>
    ...
    <more code>
}
```

Example:

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

Here `i` is a dummy variable. It's actual name doesn't matter, just its action

```
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

for-loops

Example:

```
for(n in names(iris)){
  if(is.numeric(iris[,n])){
    print(glue::glue('The mean of {n} is {mean(iris[,n])}'))
  }
}
```

```
The mean of Sepal.Length is 5.84333333333333
The mean of Sepal.Width is 3.05733333333333
The mean of Petal.Length is 3.758
The mean of Petal.Width is 1.19933333333333
```

You don't need the `<list>` in the for-loop definition to be integers. In this case it is a list of strings.

Note that vectors are also considered lists for this purpose.

The **glue** package allows you to run templated text strings interspersed with the results of R objects

purrr: functional programming and mapping

purrr

The **purrr** package provides ways to efficiently run functions over lists. These functions are typically more efficient than for-loops.

The function `purrr::map` has syntax

```
map(<list/vector>, <function/formula>, ...)
```

Example:

```
iris1 <- select(iris, where(is.numeric))
map(iris1, mean)
```

```
$Sepal.Length
[1] 5.843333

$Sepal.Width
[1] 3.057333

$Petal.Length
[1] 3.758

$Petal.Width
[1] 1.199333
```

`map` takes a list and outputs a list.

Recall, a data.frame is a list of columns, so `map` takes each column and applies the function `mean` to it, and prints the output

If you're familiar with `lapply`, `map` works almost exactly the same way

purrr

Example (cont.):

You can clean the output up a bit.

```
map_dbl(iris1, mean)
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

There are several helper functions like `map_dbl`, `map_int`, `map_chr`, and others that will reduce the output into a vector of particular type (more here)

`map` can also be used as part of pipes, leveraging the fact that data.frames are lists of columns.

```
iris %>%
  select(where(is.numeric)) %>%
  map_dbl(mean)
```

```
Sepal.Length  Sepal.Width Petal.Length  Petal.Width
    5.843333     3.057333     3.758000     1.199333
```

> **Question:** Why does `map_dbl` only have the argument `mean`?

purrr

There are several extensions of `map`

- `map2` and derivatives `map2_dbl`, etc, iterate over two lists to compute the outcome of a function of **two** variables
- `pmap` and derivatives iterate over *p* lists to compute the outcome of a p-dimensional function
- `imap` and derivatives iterates over a list and its index/names to compute the outcome of a function that takes the values and index/names as inputs

purrr

The function part of these functions can be entered in a couple of ways:

1. If you have a formal `function` f with the appropriate number of arguments, you can just add `f`.
    - `map_dbl(iris1, mean)`
2. You can also define a function "on-the-fly" using a *formula interface*.
    - `map_dbl(iris1, ~mean(.x))`
    - if you have multiple arguments, they are denoted as `.x`, `.y`, `.z`, `.w`, etc.

The second method is often referred to as *anonymous functions* or *lambda functions* in computer science since they aren't given a name