Statistical Computing with R
Masters in Data Science 503 (S5)
Fourth Batch, SMS, TU, 2025

Shital Bhandary
Associate Professor
Statistics/Bio-statistics, Demography and Medical Education
Patan Academy of Health Sciences, Lalitpur, Nepal
Faculty, Master in Medical Research, NHRC/Kathmandu University
Faculty, FAIMER Fellowship in Health Professions Education, India/USA

Review Preview

- Basics of R
- Chapter from “R for Everyone” book
- **We discussed it in the last class**

- Basics of coding in R
- Chapter from “Hands-on Programming with R” book
- **We will discuss this in today’s class**

(Variable) Naming convention is R?

- The article "**The State of Naming Conventions in R**" suggest to use:
  - alllowercase **e.g. adjustcolor**
  - period.separated **e.g. plot.new**
  - underscore_separated **e.g. numeric_version**
  - lowerCamelCase **e.g. addTaskCallback**
  - UpperCamelCase **e.g. SignatureMethod**

This link suggests to use “underscore_separated”

https://www.r-bloggers.com/2014/07/consistent-naming-conventions-in-r/

- It argues that:
  - alllowercase names are **difficult to read**, especially for non-native readers.
  - period.separated names are **confusing** for users of Python and other languages in which dots are meaningful.
  - UpperCamelBack is ***ugly*** and requires excessive use of the shift button.

Functions in R: Built-in functions

- round()
  - round(3.1415)
  - 3

round()
round(3.1415, digits = 2)
3.14

- factorial()
  - factorial(3)
  - 6
  - **3! = 3 x 2 x 1**

factorial()
factorial(2*3)
720
6! = 6 x 5 x 4 x 3 x 2 x 1

- mean()
  - mean(1:6)
  - **=(1+2+3+4+5+6)/6 = 3.5**

mean()
mean(c(1:30)
15.5

“Sample” function: Random sampling without or with replacement in R

die <- 1:6

- sample(x = die, size = 1)
- sample(x = die, size = 1)
- sample (x = die, size = 1, replace=TRUE)

- sample(x = die, size = 2)
- sample(x – die, size = 2)
- sample(x = die, size = 2, replace=TRUE)

“Sample” function to split a datafile into train and test datasets

Make sure to have “iris.csv” datafile in the working directory and use read.csv to import it in R Studio:

- read.csv(“iris.csv”)

- We can do the 70:30 random split of iris data frame as follow:
  - **set.seed(123)**
  - tt.sample <- sample(c(TRUE, FALSE), nrow(iris), replace=T, prob=c(0.7,0.3))
  - train <- iris[tt.sample,]
  - test <- iris[!tt.sample,]

This is very handy as main data is frequently divided into the Training and Testing datasets in Data Science models! **Why?**

User-defined function in R:

- my_function <- function() {}
- Where,
- my_function = name of the function e.g. roll (roll the die)

- function() = telling R that it is a user-defined function

- { = We need to start our code after this braces
- } = We need to close our codes before this braces

User-defined function 1: roll()

```
roll <- function() {
        die <- 1:6
        dice <- sample(die, size = 2, replace = TRUE)
        sum(dice)
}
```

First roll: roll()

Second roll: roll()

Third roll: roll()

Function creation in R: HOPR, Chapter 1

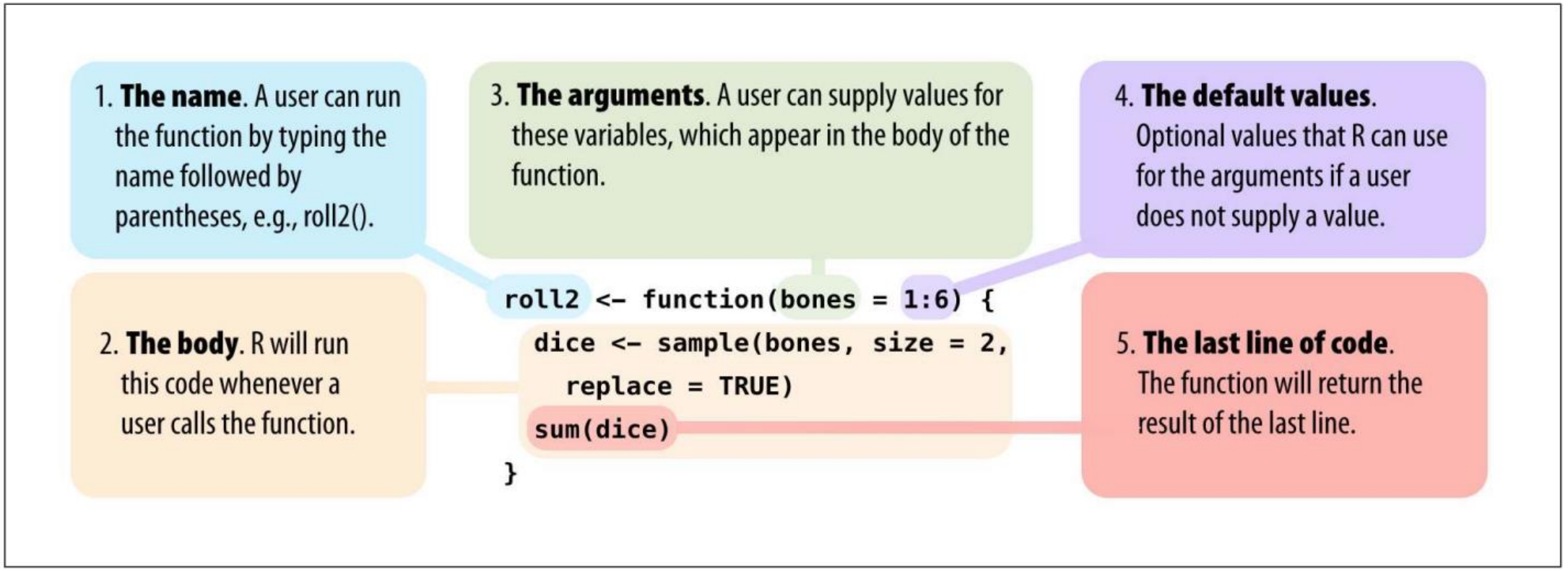

Figure 1-6. Every function in R has the same parts, and you can use function to create these parts.

User-defined function 2: roll2()

```
roll2 <- function(dice = 1:6) {
        dice <- sample(dice, size = 2, replace = TRUE)
        sum(dice)
}
```

First roll: roll2()

Second roll: roll2()

Third roll: roll2()

User-defined function 3: roll3(data?)

```
roll3 <- function(dice) {
        dice <- sample(dice, size = 2, replace = TRUE)
        sum(dice)
}
```

First roll: roll3(dice = 1:6)

Second roll: roll3(dice = 1:12)

Third roll: roll3(dice = 1:24) **# Is this possible in two dice?**

Function in R: Continued …

best_practice <- c(“Let”, “the”, “computer”, “do”, “the”, “work”)

```
print_words <- function(sentence) {
        print(sentence[1])
        print(sentence[2])
        print(sentence[3])
        print(sentence[4])
        print(sentence[5])
        print(sentence[6])
}
```

What is wrong with this approach?

- print_words(best_practice) # [1] ”Let” [1] “the” [1] “computer” [1] “do” [1] “the” [1] “work”
- print_words(best_practice[-6]) # [1] ”Let” [1] “the” [1] “computer” [1] “do” [1] “the” [1] “NA”
- best_practice[-6] # [1] ”Let” “the” “computer” “do” “the”

Can we improve it in R?
We can use functions with “for” loop in R!

```
print_words <- function(sentence) {
    for (word in sentence) {
    print(word)
    }
}
print_words(best_practice)
```

[1] “Let” [1] “the” [1] “computer” [1] “do” [1] “the” [1] “work”

```
print_words(best_practice[-6])
```

[1] “Let” [1] “the” [1] “computure” [1] “do” [1] “the”

“for” loop

R:

```
for (variable in collection) {
    do things with variable
}
```

“for” and “while” loops can be very slow in R!

What to do?

https://swcarpentry.github.io/r-novice-inflammation/03-loops-R/

Loops in R will not be slow if we:

- Don’t use a loop when a vectorized alternative exists
- Don’t grow objects (via c, cbind, etc) during the loop – R has to create new object and copy across the information just to add new element or row/column
- Allocate an object to hold the result and fill it during the loop
- **Can we do even better in R? Alternative to “loop” in R??**

While working with data.frame in R:

- It is better to use family of “apply” functions from base R:
  - apply
  - lapply
  - sapply
  - vapply

More here: https://www.datacamp.com/tutorial/r-tutorial-apply-family

We will discuss this in detail while doing breakdown analysis session in R later!

- functions instead of “for loop” to run the script much faster in R!

- Same applies to the “while loop” too!

Condition: if and else

```
if (condition) {
        #code executed when condition is TRUE
} else {
        #code executed when condition is FALSE
}
```

Can you think of an example?

What will be the output?

#Checking values of y with x:

```
if (y < 20) {
   x <- "Too low"
} else {
  x <- "Too high"
}
```

#Can you get anything from this?

#Will this work?

```
check.y <- function(y) {
if (y < 20) {
print("Too Low") } else {
print("Two high")
}}
```

- check.y(10)
- check.y(30)

Creating binary variables with “ifelse”

#Will this work?

y <- 1:40

ifelse(y<20, “Too low”, “Too high”)

It’s a logical as:

ifelse(y<20, TRUE, FALSE)

Good to make binary variables with text categories!

#Will this work?

y <- 1:40

ifelse(y<20, 1, 0)

Good to make binary variables with numerical categories!

This one is preferred in DS!

Multiple conditions:

In a function:

```
if (this) {
        # do that
} else if (that) {
        # do something else
} else if (that) {
        # do something else
} else
# remaining
}
```

```
check.x <- function(x=1:99){
if (x<20){
print("Less than 20")} else{
if (x<40) {
print("20-39")} else{
if (x<100) {
print("41-99")}
}}}
```

- check.x(15)
- check.x(30)
- check.x(45)

Good to make categorical variables!

Multiple Conditions: combining “ifelse”

Will this work too?

```
x <- 1:99
x1 <- ifelse(x<20, 1,0) #Binary numbers
x2.1 <- ifelse(x<20, “<20”, “20+”) #Binary text
```

x2.2 ? For x between 20 and less than 40

x2.3 ? For x between 40 and less than 100

Now combine them in a single column with <20 =1, 20-39 = 2 and 40-99 = 3 for x i.e. create categorical variable of x!

Will this work?

```
x3 <- ifelse(x<20,1,ifelse(x<40,2,3))
x3
table(x3)
```

```
#This code shows how Petal. Length categories was created from Petal. Length variable of iris data frame
iris <- within(iris, {
Petal.cat <- NA
Petal.cat[Petal.Length <1.6] <- "Small"
Petal.cat[Petal.Length >=1.6 & Petal.Length<5.1] <- "Medium"
Petal.cat[Petal.Length >=5.1] <- "Large"
})
```

#The 1.6=Q1 and 5.1=Q3 were obtained from the “summary” of the Petal.Lenght variable i.e. summary(iris$Petal.Length)

```
Iris$Petal.cat
table(iris$Petal.cat)
```

Multiple Conditions: If, else if, else if, else if

```
#Make this function work!
if (temp <= 0) {
"freezing"}
else if (temp <= 10) {
"cold"}
else if (temp <= 20) {
"cool"}
else if (temp <= 30) {
"warm"}
else {
"hot"}
```

```
# What is missing?

# How to address it?
```

Questions/queries?

Thank you!

@shitalbhandary