Lecture 3: Functions and control flow

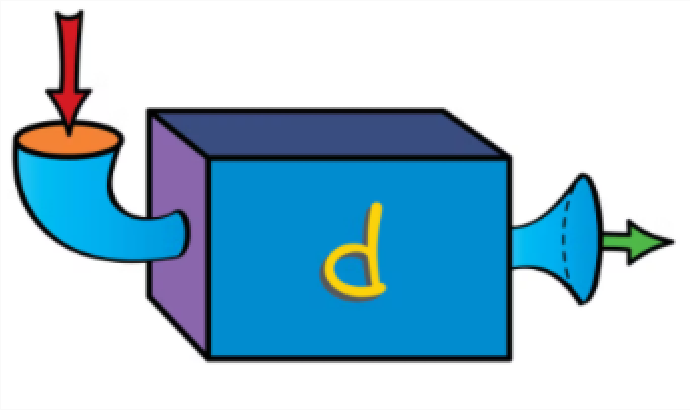
The heart of any programming language

Ingmar Sturm
UCSB
2024-06-26

Special thanks to Robin Liu for select course content used with permission.

Functions

What is a function?



img credit: Robert Ghrist

This function, called **d**, takes some inputs (red arrow) and **returns** some output (green arrow).

The inputs to a function are also called **arguments**.

Functions

Anatomy of a function in R

```
increment_power <- function(x, pwr = 2) {
  x <- x + 1
  return(x^pwr)
}
increment_power(2, 3)
```

```
## [1] 27
```

```
increment_power(5)
```

```
## [1] 36
```

`increment_power` is the *name* of this function. It has two *arguments*. The *body* of `increment_power` are the two lines below. It *returns* the value of `x^pwr`.

```
x <- x + 1
return(x^pwr)
```

Functions

Anatomy of a function in R

```r
increment_power <- function(x, pwr = 2) {
  x <- x + 1
  return(x^pwr)
}
increment_power(2, 3)
```

```
## [1] 27
```

```r
increment_power(5)
```

```
## [1] 36
```

`pwr` is a **default** argument with default value 2. You see default arguments everywhere in the R help:

```r
matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
       dimnames = NULL)
```

Functions

Anatomy of a function in R

```r
increment_power <- function(x, pwr = 2) {
  x <- x + 1
  return(x^pwr)
}
increment_power(x = 2, pwr = 3)
```

```
## [1] 27
```

```r
increment_power(x = 5)
```

```
## [1] 36
```

Specifying the name of the argument often improves code readability.

Last class, we specified these names:

```r
x <- matrix(1:6, nrow = 2, ncol = 3)
```

Default arguments

Another example from the R help.

```
## Default S3 method:
seq(from = 1, to = 1, by = ((to - from)/(length.out - 1)),
    length.out = NULL, along.with = NULL, ...)
```

```
seq(1, 4, 0.5)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0
```

```
seq(1, 4, by = 0.5)
```

```
## [1] 1.0 1.5 2.0 2.5 3.0 3.5 4.0
```

Remember a main goal of coding is to communicate with other coders.

Functions

```r
increment_power <- function(x, pwr = 2) {
  x <- x + 1
  return(x^pwr)
}
```

1. Implement the `increment_power` function
2. What happens if you pass in a vector for `x`? For `pwr`? For both?
3. Is this function vectorized? Did recycling occur?

You should always test your function on a variety of different inputs. What is the result of `increment_power("cat")`?

03:00

Functions

The value of the last statement in a function is automatically returned.

This is a quirk if you're used to other languages.

```
increment_power <- function(x, pwr = 2) {
  x <- x + 1
  x^pwr # No "return" keyword!
}
increment_power(2, 3)
```

```
## [1] 27
```

Branching

if, else, ifelse

Branching

We often want to decide what to do based on the truth-value of a logical expression.

```
logical_exp <- FALSE
if (logical_exp) {
  print("It is true")
} else {
  print("It is false")
}
```

```
## [1] "It is false"
```

Most useful in functions and loops.

Branching

What is the result of the following?

```r
logical_exp1 <- TRUE
logical_exp2 <- TRUE
if (logical_exp1) {
  print("A")
} else if (logical_exp2) {
  print("B")
} else {
  print("C")
}
```

What about this one?

```r
logical_exp1 <- TRUE
logical_exp2 <- TRUE
if (logical_exp1) {
  print("A")
}
if (logical_exp2) {
  print("B")
} else {
  print("C")
}
```

00:30

Branching in a function

How I will assign grades for the class

```r
grade <- function(x) {
  if (x > 90) {
    "A"
  } else if (x > 80) {
    "B"
  } else if (x > 50) {
    "C"
  } else {
    "F"
  }
}
```

```r
grade("89.999")
```

```
## [1] "B"
```

jk

Branching

Create a function `noise` that takes a farm animal and returns its sound. Possible animals are cow, pig, dog, and owl.

```r
noise("cow")
```

```
## [1] "moo"
```

```r
noise("pig")
```

```
## [1] "oink"
```

```r
noise("owl")
```

```
## [1] "hoot"
```

```r
noise("dog")
```

```
## [1] "woof"
```

```r
noise("capybara")
```

```
## [1] "Animal is not recognized!"
```

05:00

Branching

Create a function `parity` that takes a number and returns whether its even or odd. If neither even or odd (e.g. input is a decimal), return "Not an integer!".

Hint: the `%%` operator is the **modulus** operator. `x %% y` gives the remainder when `x` is divided by `y`.

```
parity(4)
```

```
## [1] "even"
```

```
parity(17)
```

```
## [1] "odd"
```

```
parity(12.43)
```

```
## [1] "Not an integer!"
```

05:00

Loops

For loop

Typical scenario: loop over a *vector* of stuff.

```r
for (animal in c("cow", "pig", "dog", "owl")) {
  print(paste(animal, "says", noise(animal)))
}
```

```
## [1] "cow says moo"
## [1] "pig says oink"
## [1] "dog says woof"
## [1] "owl says hoot"
```

For loop

There are equivalent ways to do the previous loop.

```
animals <- c("cow", "pig", "dog", "owl")
seq_along(animals)
```

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

```
for(i in seq_along(animals)) {
  print(paste(animals[i], "says", noise(animals[i])))
}
```

```
## [1] "cow says moo"
## [1] "pig says oink"
## [1] "dog says woof"
## [1] "owl says hoot"
```

Now we have the loop *index* `i` we can use for other stuff.

The letters `i`, `j`, and `k` are typically used as the loop index variable. Follow this!

For loop

Modify the previous loop to create a vector of animal sounds.

```r
animals <- c("cow", "pig", "dog", "owl")
noises <- vector(length = length(animals)) # an "initialized" vector to fill out
for(i in seq_along(animals)) {
  noises[i] <- noise(animals[i])
}
noises
```

```
## [1] "moo"  "oink" "woof" "hoot"
```

We needed a loop index to fill out the result vector.

Function with branching loop

Create a function `parity_vec` that takes a vector and returns a vector of "even" or "odd" depending on the corresponding entry.

```
parity_vec(1:5)
```

```
## [1] "odd"  "even" "odd"  "even" "odd"
```

```
parity_vec(c(12, 320, 598, 23))
```

```
## [1] "even" "even" "even" "odd"
```

05:00

Vectorized ifelse

```
ifelse(c(T, F, T, T), "hello", "goodbye")
```

```
## [1] "hello"   "goodbye" "hello"   "hello"
```

Here is an example of when a vectorized solution beats using a loop. Rewrite `parity_vec` to use the vectorized `ifelse`.

Hint: The modulus operator `%%` is vectorized: run `1:10 %% 2` in the console.

```
parity_vec(1:5)
```

```
## [1] "odd"  "even" "odd"  "even" "odd"
```

```
parity_vec(c(12, 320, 598, 23))
```

```
## [1] "even" "even" "even" "odd"
```

Now our function is **vectorized**.

It wasn't before because we used a loop.

02:00

sapply

A common operation

Recall our `noise` function. What if we wanted to pass in a *vector* of animals and return a *vector* of sounds?

```
> noise(c("cow", "owl"))
Error in if (animal == "cow") { : the condition has length > 1
```

`noise` is not a vectorized function

```
sapply(c("cow", "owl"), noise)
```

```
##    cow    owl
##  "moo" "hoot"
```

Behind the scenes `sapply` creates a loop and applies `noise` to each element of `c("cow", "owl")`, returning a vector of results

sapply

`sapply` simplifies this kind of operation:

```
animals <- c("cow", "pig", "dog", "owl")
noises <- vector(length = length(animals))
for(i in seq_along(animals)) {
  noises[i] <- noise(animals[i])
}
noises
```

```
## [1] "moo"  "oink" "woof" "hoot"
```

```
sapply(animals, noise)
```

```
##    cow    pig    dog    owl
##  "moo" "oink" "woof" "hoot"
```

sapply

Do we always need it?

Not if our function is already vectorized!

```
increment_power <- function(x, pwr = 2) {
  x <- x + 1
  return(x^pwr)
}
sapply(1:4, increment_power)
```

```
## [1]  4  9 16 25
```

```
increment_power(1:4) # faster since no loop is created
```

```
## [1]  4  9 16 25
```

Not covered

I did not mention `while` and `break` in this class. Here it is finally:

```r
i <- 1
while(TRUE) {
  if (i == 4) {
    break
  }
  print("this takes a while")
  i <- i + 1
}
```

```
## [1] "this takes a while"
## [1] "this takes a while"
## [1] "this takes a while"
```

This construct is common, but I didn't want to clutter this lecture. It turns out that `for` loops are generally better than `while` loops.

Problem solving

We now have enough tools to solve some basic programming puzzles.

Next lecture, we will explore some leetcode problems with R.

Main tools: vectors, functions, branching, loops.

Problem solving example

You are given a numeric vector `v` and a number `target`. Create a function `remove_elt(v, target)` that returns a vector containing elements of `v` but with `target` removed.

```
remove_elt <- function(v, target) {
  # Your code here
}
```

```
remove_elt(c(2, 3, 3, 5), 3)
```

```
## [1] 2 5
```

```
remove_elt(c(14, 14, 7, 7, 14, 10), 14)
```

```
## [1]  7  7 10
```

Main takeaways

There was a lot of info today...

- Functions let you perform the same operation on a variety of different inputs
- Branching and looping are useful ideas within a function.
- Always look for vectorization before reaching for a loop.
- Passing in a vector to a function is important. If function is not vectorized, consider `sapply`, or more advanced techniques (maybe more later).
- You must practice this **a lot**.