Iterations and Loops

Programming Structures

Gaston Sanchez

STAT 33B, Fall 2025

Iterative constructs

- Many times we need to perform a procedure **several times**
- The main idea is that of **iteration**
- For this purpose we use loops
- R provides three basic iterative paradigms:
  - `for`
  - `while`
  - `repeat`

Big Favaor

In order to describe some of the concepts around R loops, I'm going to ask you to forget about vectorization.

For illustration purposes, I'll describe some operations "manually", step by step.

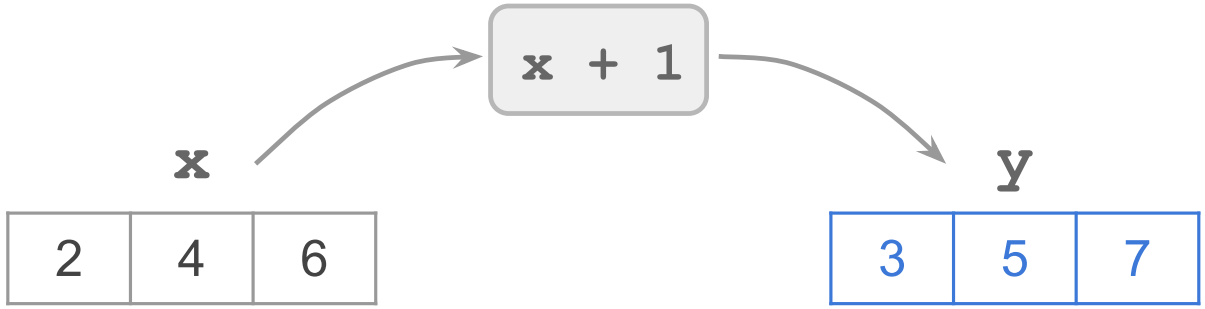
For Loops

Loops

Often we want to repeatedly carry out some computation a fixed number of times.

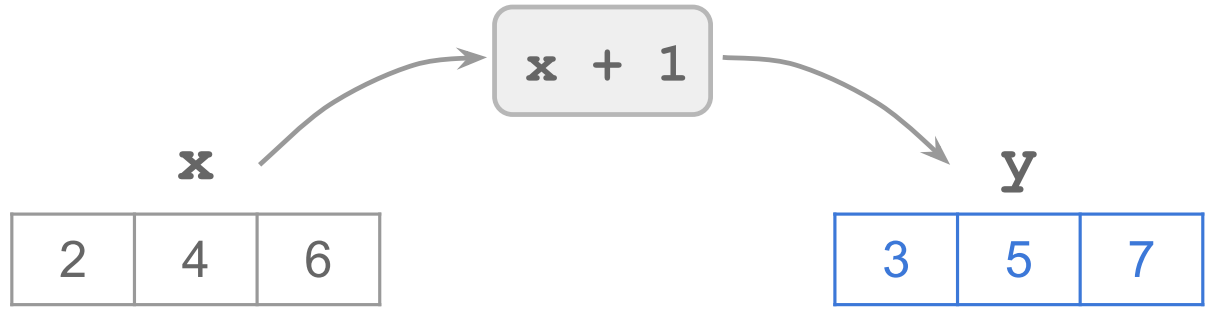
For instance, repeat an operation for each element of a vector. In R this is done with a `for()` loop.

Toy example of `for()` loop



*How to get **x+1**, step-by-step, without using vectorization?*

Toy example of `for()` loop




```
# initialize empty y
y <- rep(0, 3)

y[1] <- x[1] + 1
y[2] <- x[2] + 1
y[3] <- x[3] + 1
```

Toy example of `for()` loop


```
# initialize empty y
y <- rep(0, 3)

y[1] <- x[1] + 1
y[2] <- x[2] + 1
y[3] <- x[3] + 1
```

They all have the same format:

```
y[pos] <- x[pos] + 1
```

(`pos` *indicates position)*

Anatomy of `for()` loop

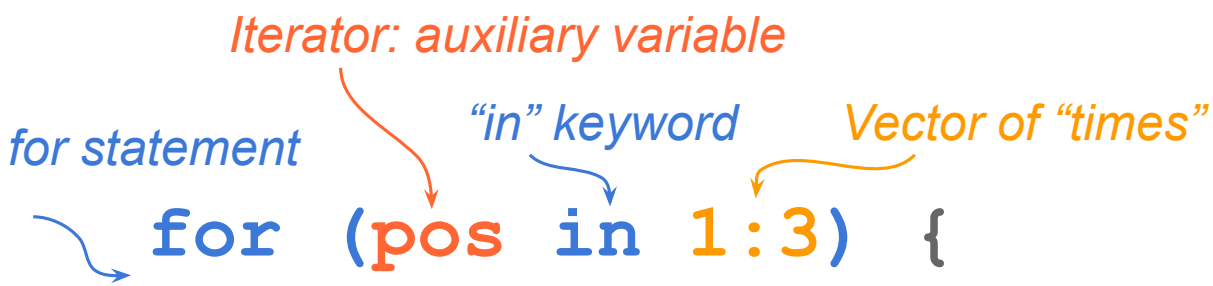
```
x <- c(2, 4, 6)
y <- rep(0, 3)
```

```
for (pos in 1:3) {
  y[pos] <- x[pos] + 1
}
```

Anatomy of `for()` loop

```
x <- c(2, 4, 6)
y <- rep(0, 3)

for (pos in 1:3) {
  y[pos] <- x[pos] + 1
}
```



For Loop

A for-loop runs a block of code once for each element of a vector or list:

```
x <- c(-15, 12, 3)

for (elt in x) {
  message("The element is ", elt)
}
```

```
## The element is -15

## The element is 12

## The element is 3
```

For Loop

- The idea is the same as for-loops in other languages.
- Notice that you don't declare the iterator outside the loop.
- Also, in most cases you don't explicitly increase the iterator.

For Loop

```r
x <- c(-15, 12, 3)

for (i in 1:3) {
  elt = x[i]
  message("The element at position ", i, " is ", elt)
}
```

```
## The element at position 1 is -15

## The element at position 2 is 12

## The element at position 3 is 3
```

Curly braces { } are only required if you have multiple lines of code.

For Loop and the break statement

Use break to exit a loop early:

```
x <- c(-15, 12, 3)

for (elt in x) {
  if (elt %% 2 == 0)
    break

  message("The element is ", elt)
}
```

```
## The element is -15
```

For Loop and the `next` statement

Use `next` to skip to the next iteration early:

```
x <- c(-15, 12, 3)

for (elt in x) {
  if (elt %% 2 == 0)
    next

  message("The element is ", elt)
}
```

```
## The element is -15
```

```
## The element is 3
```

Loop indices

If you need indices, using `1:n` can cause bugs:

```
n = 0
for (i in 1:n) {
  message(i)
}
```

```
## 1
```

```
## 0
```

Loop indices

If you need indices, use `seq_len(n)` instead of `1:n`

```
n = 0
for (i in seq_len(n)) {
  message(i)
}
```

Loop indices

Similarly, using `1:length(x)` can cause bugs:

```
#x = c(-3, 5, 7)
x = c()
for (i in 1:length(x)) {
  message("The element is ", x[i])
}
```

```
## The element is
## The element is
```

Loop indices

Use `seq_along(x)` instead:

```
# x = c(-3, 5, 7)
x = c()
for (i in seq_along(x)) {
  message("The element is ", x[i])
}
```

More generally, use `seq()` to produce sequences of indices.

Loop indices

The vector of `times` does not have to be a numeric vector; it can be any vector

```
value <- 2
times <- c('1', '2', '3', '4', '5')

for (i in times) {
  value <- value * 2
  print(value)
}
```

```
## [1] 4
## [1] 8
## [1] 16
## [1] 32
## [1] 64
```

While Loops

While Loop

A while-loop runs a block of code repeatedly while some condition is `TRUE`. The condition is checked before each iteration:

```
even <- seq(0, 50, 2)
total <- 0
i <- 1

while (total < 50) {
  total <- total + even[i]
  i <- i + 1
}

total
```

```
## [1] 56
```

```
i
```

```
## [1] 9
```

Repeat Loops

Repeat Loop

Some languages have a *do-while-loop*, which checks the condition **after** each iteration (so the first iteration always runs).

R has `repeat`, which is the same as `while (TRUE)`.

You can create a do-while-loop in R with `repeat`

```
total <- 20

repeat {
  total <- total * 2
  if (total > 10)
    break
}
total
```

```
## [1] 40
```

Repeat Loop

- `repeat` loops are like "reverse" `while` loops
- in `repeat` loops you execute some code and then check a stopping condition
- computations are carried out for as long as the condition is FALSE
- the loop stops when the condition is TRUE
- If you enter an infinite loop, break it by pressing `ESC` key

Preallocation and Iteration Strategies

For Loops and Vectorized Computations

- R loops have a bad reputation for being slow
- Experienced useRs will tell you to avoid loops in R
- It is not really that the loops are slow; the slowness has more to do with what you do inside the loops
- A typical source of slowness has to do with the way R handles the boxing and unboxing of data objects, which may be a bit inefficient.

For Loops and Vectorized Computations

- When using R, you may need to start solving a problem using a loop. Once you solved it, try to see if you can find a vectorized alternative.
- It takes practice and experience to find alternative solutions to R loops.

Preallocation

Preallocation means allocating memory for results before a computation.

These functions allocate vectors:

- `character()`
- `complex()`
- `numeric()`
- `logical()`
- `vector()`
- `rep()`

Preallocation

Examples:

```
character(3)
complex(10)
numeric(4)
logical(3)
vector("logical", 6)
vector("list", 3)
```

Preallocation in loops

Preallocation is especially important for loops:

```
# BAD, NO PREALLOCATION:
x <- c()
for (i in 1:10000) {
  x <- c(x, i * 2)
}
```

This example is extremely inefficient because `x` "grows" at every iteration

Preallocation in loops

Preallocation is especially important for loops:

```
# GOOD: WITH PREALLOCATION
n <- 10000
x <- numeric(n)
for (i in seq_len(n)) {
  x[i] <- i * 2
}
```

Compared to the previous slide, the above code is more efficient because we have preallocated `x` with the right "size"

Developing Iterative Code

When thinking about writing a loop, try (in order):

1. vectorization
2. apply functions
   - Try an apply function if iterations are independent.
3. for/while-loops
   - Try a for-loop if some iterations depend on others.
   - Try a while-loop if the number of iterations is unknown.
4. recursion
   - Convenient for naturally recursive problems (like Fibonacci), but often there are faster solutions.

Developing Iterative Code

- Before you write the loop, try writing the code for just 1 iteration.
- Make sure that code works; it's easy to test code for 1 iteration.
- When you have 1 iteration working, then try using the code in a loop (you will have to make some small changes).
- If your loop doesn't work, try to figure out which iteration is causing the problem. One way to do this is to use `message()` to print out information.
- Then try to write the code for the broken iteration, get that iteration working, and repeat this whole process.

Choosing an Iteration Strategy