

# R: `for` is vector controlled

R's `for` structure is actually **vector controlled repetition**, a special case of counter controlled repetition

```
for ( var in seq ) {  
  expression  
}
```

*seq is an expression that evaluates to a vector*

*var will in turn be assigned the value of each element in the vector*

# R: `for` is vector controlled

## Example

```
a <- c(0.51, 0.57, 0.09, 1.02, 1.10)
for ( number in a ) {
  print(number * 2)
}
```

**var**

**seq**

loop 4

0.51
0.57
0.09
1.02
1.10

**a**

“number” is in turn  
each element of a

# R: `for` is vector controlled

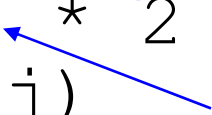
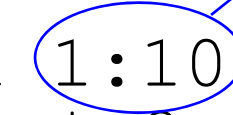
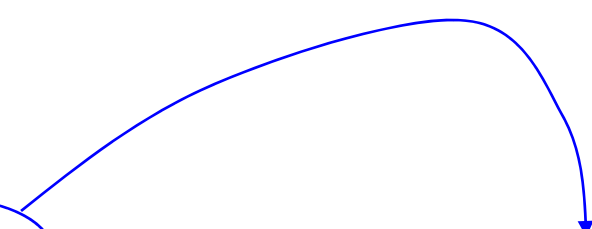
## Example revisited

```
for ( i in 1:10 ) {  
  j <- i * 2  
  print(j)  
}
```

*i* is in turn  
each  
element of  
the vector

1
2
3
4
5
6
7
8
9
10

vector



# Vector controlled repetition

Vector controlled repetition is a **special case** of counter controlled repetition

```
v #vector
n <- length(v)
i <- 1
while ( i <= n ) {
  expression using v[i]
  i <- i + 1
}
```

get vector length

index vector elements

optional

# Vector controlled repetition

- Many languages have convenience structures for vector (or container) controlled repetition
- Often called **foreach** or similar
- General pseudocode:

for each item in container  
do something

# R: vector control with lists

## List

- special type of vector
- container for multiple objects

```
mylist <- list(obj1, obj2, obj3)
for ( object in mylist ) {
  expression
}
```

creates a list

could do something to or with the object (or not)

# R: vector control with lists

## Example

a, b, c, d  
are numerical  
vectors

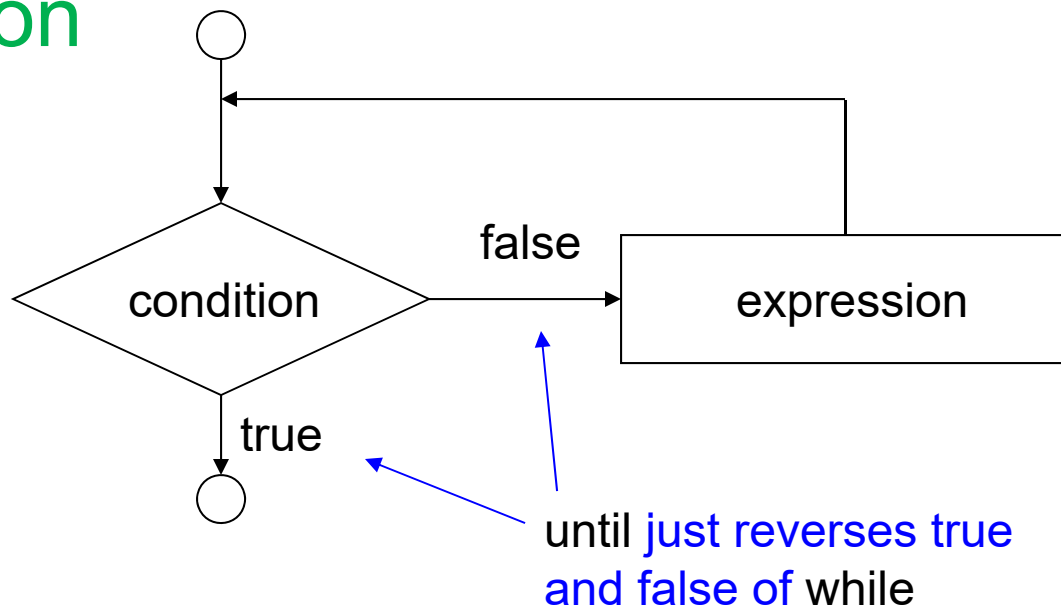
```
datasets <- list(a,b,c,d)
for ( x in datasets ) {
  hist(x)
}
```

What does this do?

# until sentinel control

- Many languages have an **until** structure
- General (non-R) syntax:

```
until ( condition ) {  
    expression  
}
```



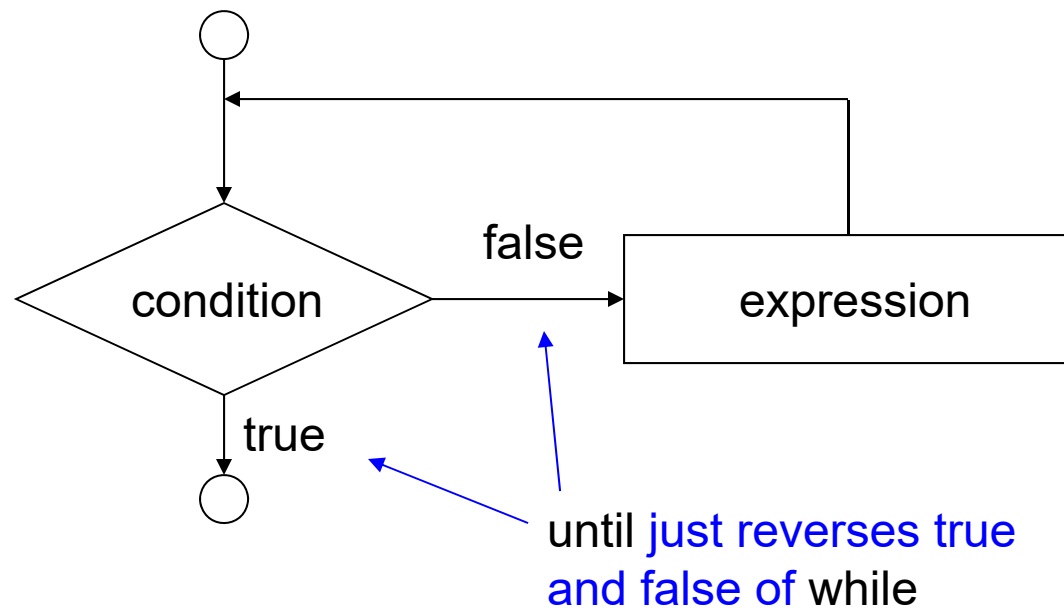


# until sentinel control

- In R:

```
while ( !condition ) {  
    expression  
}
```

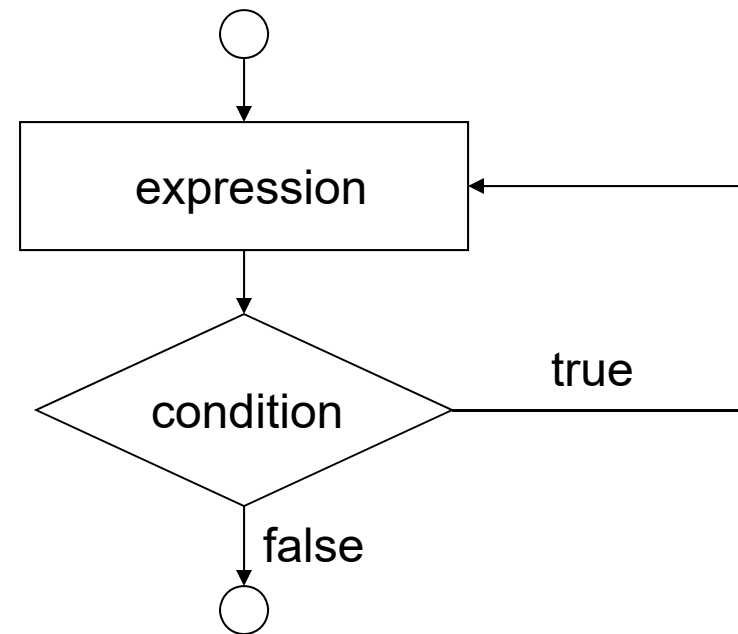
NOT operator  
see ?Logic



# do-while sentinel control

- Do action **at least once** before evaluating the condition
- General (non-R) syntax:

```
do {  
    expression  
} while ( condition )
```



# do-while sentinel control

- How to do this in R

## Option 1

```
expression  
while ( condition ) {  
    expression  
}
```

Advantage: clear expression is evaluated at least once

Disadvantage: duplicate expression (esp. multiple lines)

# do-while sentinel control

- How to do this in R

Option 2 – use a flag

```
first <- TRUE
while ( condition | first ) {
  expression
  if ( first ) first <- FALSE
}
```

Advantage: uses a proper while structure (“goto”-less)

Disadvantage: long and clunky, slight performance hit

# do-while sentinel control

- How to do this in R

Option 3 – goto style

```
while ( TRUE ) {  
    expression  
    if ( !condition ) break  
}
```

infinite loop

“break out of the while structure”

Disadvantages: - unintuitive while statement  
- break is goto style

# do-while sentinel control

- The recommended way in R

Option 4 – R's `repeat` repetition structure

```
repeat {  
    expression  
    if ( !condition ) break  
}
```

Advantage: clear, concise, no performance hit

Disadvantage: goto style; `repeat` is not universal

# goto keywords

- Use sparingly and with care
  - `break`
  - `next`
- see ?Control
- Sometimes useful
  - performance
  - readability (to avoid `if` nested in a loop)

# break

## goto style

```
for ( i in 1:n ) {  
    expression 1  
    if ( condition ) break  
    expression 2  
}
```

Advantage: clearer if code is short  
and **break** stands out.

Disadvantage: we don't expect  
repetition structures to exit early.

## structured style for

```
done <- FALSE  
for ( i in 1:n ) {  
    if ( !done ) {  
        expression 1  
        if ( condition ) {  
            done <- TRUE  
        } else {  
            expression 2  
        }  
    }  
}
```

Disadvantages: long; deep nested  
expressions; **for** does all n



# break

## goto style

```
for ( i in 1:n ) {  
  expression 1  
  if ( condition ) break  
  expression 2  
}
```

Advantage: clearer if code is short  
and **break** stands out.

Disadvantage: we don't expect  
repetition structures to exit early.

## structured style while

```
done <- FALSE  
i <- 1  
while ( !done | i <= n ) {  
  expression 1  
  if ( condition ) {  
    done <- TRUE  
  } else {  
    expression 2  
  }  
  i <- i + 1  
}
```

Disadvantages: long; nested  
expression 2

# next (or non-R continue)

## goto style

```
for ( i in 1:n ) {  
    expression 1  
    if ( condition ) next  
    expression 2  
}
```

Advantage: expression 2 is not nested.

Disadvantage: we don't expect repetition structures to exit early.

## structured style for

```
for ( i in 1:n ) {  
    expression 1  
    if ( !condition ) {  
        expression 2  
    }  
}
```

Advantage: structured, still clear.

Disadvantage: hardly any; nested expression 2.

# Theory: **while** is fundamental

All repetition structures can be built from **while**

## Fundamental

```
while (condition) {  
  expression  
}
```

## repeat

```
while (TRUE) {  
  expression  
}
```

## until

```
while (!condition) {  
  expression  
}
```

## Counter control

```
n #Number of reps  
i <- 1  
while (i <= n) {  
  expression  
  i <- i + 1  
}
```

## Vector control

```
v #A vector  
n <- length(v)  
i <- 1  
while (i <= n) {  
  expression on v[i]  
  i <- i + 1  
}
```

## do-while

```
first <- TRUE  
while (condition | first) {  
  expression  
  if (first) first <- FALSE  
}
```

# R repetition structures in practice

## while sentinel control

```
while ( condition ) {  
  expression  
}
```

## for counter control

```
for ( i in 1:n ) {  
  expression  
}
```

## until sentinel control

```
while ( !condition ) {  
  expression  
}
```

## foreach vector control

```
for ( element in vector ) {  
  expression  
}
```

## do-while sentinel control (e.g. option 4)

```
repeat {  
  expression  
  if ( !condition ) break  
}
```

## foreach vector control with lists

```
for ( object in list ) {  
  expression  
}
```