Flow control structures II

Yufeng Huang

Associate Professor of Marketing, Simon Business School

August 9/11, 2022

Today

- Repetiton structures
  - for (i in vector/list) { do something }
  - while (condition is TRUE) { do something }
  - repeat { do something ; if (some exit condition) break }
- Examples
  - how to repeatedly add up life time value?
  - how to load many datasets?
  - how to calculate profit-maximization problem using loop?
- Speed issues

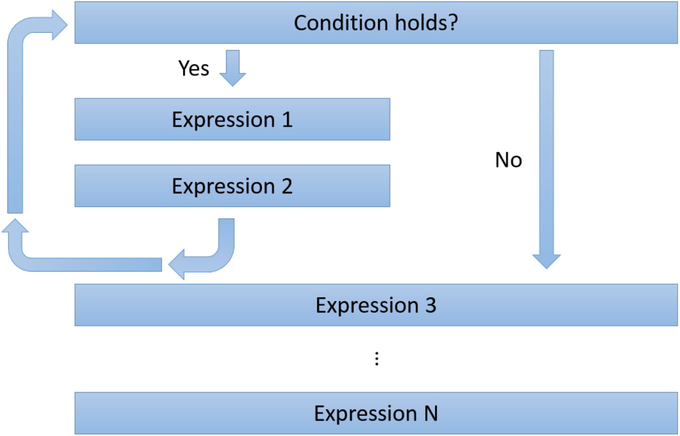
Recall my meal choice

```
# copy and paste everything?

d = 51
if (d %% 2 == 1) {
        possible_actions[1]
} else {
        possible_actions[2]
}

d = 52
if (d %% 2 == 1) {
        possible_actions[1]
} else {
        possible_actions[2]
}

d = 53
if (d %% 2 == 1) {
        possible_actions[1]
} else {
        possible_actions[2]
}

d = 54
if (d %% 2 == 1) {
        possible_actions[1]
} else {
        possible_actions[2]
}

# And so on...
```

Can write a loop on this

```
# loop version

d <- 51
while (d <= 58) {
        if (d %% 2 == 1) {
                print(possible_actions[1])
        } else {
                print(possible_actions[2])
        }
        d <- d + 1
}

# reads: day is initially 51
#    as long as day do not exceed 58
#    execute the same decision rule
# and advance one more day
```

Graphically



Repetition statements

Why loop?

- Often there are sections of code that is repeated
- Copy paste?
  - do-able for $<$10 replications
  - hard to change
  - easy to make mistakes
  - makes the code long and un-readable
  - how about 10000 replications?
- A loop statement repeatedly executes compound expressions under certain conditions

Example problem where we need a loop

- Calculate:
  - number of times a consumer has bought a product in the past
  - highest price at which the consumer has purchased the product
- Data looks like

id	month	price	buy	times_bought	max_accepted_price
1	1	2	1	NA	NA
1	2	1.8	0	1	2
1	3	1.9	1	1	2
1	4	1.6	0	2	2
2	1	2	0	NA	NA
2	2	1.8	0	0	NA
2	3	1.9	1	0	NA
2	4	1.7	0	1	1.9

- Important because we use measures similar to these to figure out which customers are more valuable than others
  - but how do we get these?

10 years ago, I wrote a pyramid when I could use a flow control structure

While loops

While loops

- "While" loop executes a group of expressions repeatedly given a condition is met
- The loop exits once the condition is not met

```
# while loop structure
while (condition) {
        keep_doing_this
}
```

Example: print and increase i as long as it's below 3

```
# print and increase i as long as it's below 3
i <- 1
while (i < 3) {
        print(i)
        i <- i + 1
}

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

Break it down

```
# first initiate i
i <- 1
# i is 1

#=====ROUND 1======
# checks condition i < 3
i < 3

## [1] TRUE

# TRUE; execute print(i); and then replace i with i + 1
print(i)

## [1] 1

i <- i + 1
# i is now 2
```

Break it down

```
#=====ROUND 2======
# checks condition i < 3
i < 3

## [1] TRUE

# TRUE; execute print(i); and then replace i with i + 1
print(i)

## [1] 2

i <- i + 1
# i is now 3

#=====ROUND 3======
# checks condition i < 3
i < 3

## [1] FALSE

# FALSE; exit
# nothing else happens
```

Example 1: customer lifetime value (CLV)

- I initially have acquired 1,000 customers
  - 5% drops out every month
  - the remaining customers find my service less “new” and are therefore willing to spend less
- Question:
  - what is the expected lifetime earning I can acquire from each of them
  - knowing this, what is the maximum marketing cost I am willing to pay to acquire these customers?

CLV

- CLV as the discounted sum of monthly profit from a customer

$$\begin{aligned} CLV_i &= \pi_{i0} + \delta\pi_{i1} + \delta^2\pi_{i2} + \ldots \\ &= \sum_{t=0,1,\ldots} \delta^t \pi_{it} \end{aligned}$$

with monthly discount factor $\delta = \frac{r}{1+d}$ where
    - $r = 0.95$ is retention rate
    - $d = 0.10$ is the monthly cost of capital[1]

[1] Company's time value between earnings today versus tomorrow; implicitly whatever I did not earn today I could borrow from the market with cost d

Recursive CLV

- Profit is not constant, e.g

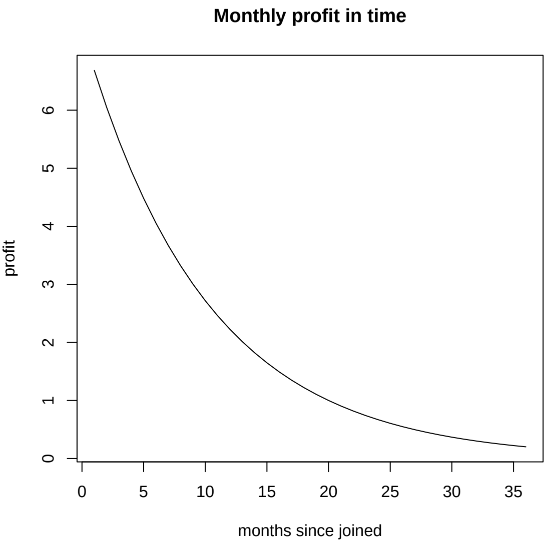
$$\pi_{it} = \exp\left(2 - 0.1 \cdot t\right)$$

  which is positive but goes to zero
- Can define CLV recursively as

$$CLV_{it} = CLV_{it-1} + \delta^t \pi_{it}$$

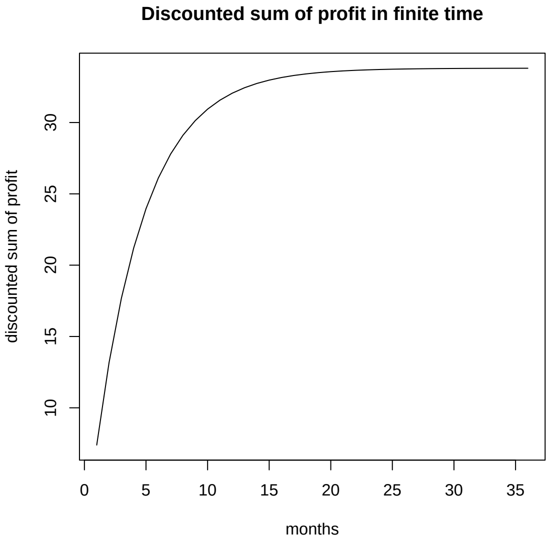
  in some sense this is "finite time value"
- Let's try approximating CLV in finite time, using a loop

```r
# compute profit for T = 36 (3 year)
profit <- exp(2 - 0.1 * 1:36)   # profit
plot(profit, type = 'l', ylab = "profit",
        xlab = "months since joined", main = "Monthly profit in time")
```



How to compute CLV?

```
# one method is to loop over time (let's do it for T = 100)

# initialize variables; important!
delta <- (1 - 0.05) / (1 + 0.1) # discount factor
CLV <- 0         # initialize CLV
t <- 0           # initialize time

# write a while loop to sum up CLV recursively
while (t <= 100) {
    CLV <- CLV + delta^t * exp(2 - 0.1*t)
    t <- t + 1
}

# check answer
CLV

## [1] 33.80953
```


Discounted sum of profit in finite time
discounted sum of profit
10
15
20
25
30
0
5
10
15
20
25
30
35
months

General thoughts on how to write a loop

- Let's recall why we need a repetition structure
  - we have sections of code that are repeated in pattern
  - i.e. similar, but not entirely the same code, is repeated
- In the above example, the recursive structure defines the repetition pattern:

  $$CLV_{it} = CLV_{it-1} + \delta^t \cdot \exp\left(2 - 0.1 \cdot t\right)$$

  and by observing this repetition pattern, we have implicitly three things to do:
  - what are the variable that change in each step?
    - 1) t, which takes integers from 1 to T
    - 2) CLV, which replaces itself for each t = 1, ..., T
  - what is the pattern (section of code) that is repeated?
    - CLV <- CLV + delta ^ t * exp(2 - 0.1 * t)
    - (but of course do not forget that in a while loop, we need t <- t + 1 at the end to advance t)

For loops

For loops

- "For" loop executes a series of statement by defining an iterator that changes over the range of integers
- Exits when the range is exhausted

```
# for loop structure
for (iterator in times) {
    do_something
}
```

Familiar example: print and increase i as long as it is below 3

```r
# print and increase i as long as it's below 3
for (i in 1:2) {
    print(i)
}
```

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

Note that the vector in *times* does not have to be numeric

```r
# define a character vector
v <- c('once', 'twice')

# can loop over character
for (i in v) {
    print(i)
}

## [1] "once"
## [1] "twice"

# but the body of the loop must be well-defined
for (i in v) {
    print(i^2)
}

        ## Error in i^2:  non-numeric argument to binary operator
```

Example 2: plot the piece-wise linear demand curve without ifelse()

- Recall that we had the piece-wise linear demand curve

$$sales = \begin{cases} 4 - 2 \cdot price & \text{if } price > 1 \\ 5 - 3 \cdot price & \text{if } price \leq 1 \end{cases}$$

- How can we compute the sales vector and plot it, without using ifelse()

```r
# price is a vector of length 20
price <- seq(0.5, 1.5, length.out = 50)

# but we can't apply the if-else statement to a vector
if (price > 1) {
        sales <- 4 - 2*price
} else {
        sales <- 5 - 3*price
}
```

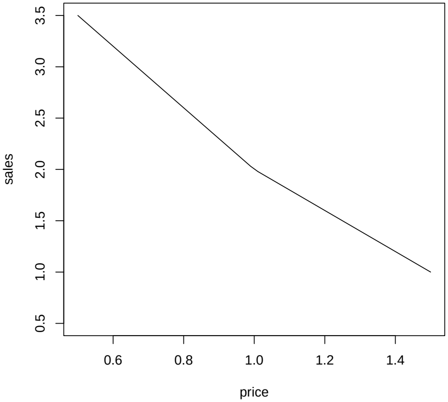
```
## Error in if (price > 1) {:  the condition has length > 1
```

How to write it as a loop?

```
# The NUMBER ONE step is to figure out how each iteration runs
# i.e., write down an example
i <- 1
price[i]        # price

## [1] 0.5

# sales[i]
sales <- rep(NA, length(price)) # pre-assign
if (price[i] > 1) {
        sales[i] <- 4 - 2*price[i]
} else {
        sales[i] <- 5 - 3*price[i]
}
sales[i]        # sales for this particular price

## [1] 3.5
```

```r
# If the example works, now expand it to all i's

# pre-assign sales
sales <- rep(NA, length(price))
# loop
for (i in 1:length(price)) {
        if (price[i] > 1) {
                sales[i] <- 4 - 2*price[i]
        } else {
                sales[i] <- 5 - 3*price[i]
        }
}

head(price)

## [1] 0.5000000 0.5204082 0.5408163 0.5612245 0.5816327 0.6020408

head(sales)

## [1] 3.500000 3.438776 3.377551 3.316327 3.255102 3.193878
```

```r
# plot the demand curve (maintain the same axis)
plot(price, sales, ylim = c(0.5, 3.5), type = 'l')
```



Example 3: reading the Billboard data

- Question: which part of the code is repeated?

```r
# Let's read Billboard data

# define url
url1 <- 'https://dl.dropboxusercontent.com/s/0r7gcf9gxk2nm4d/artists.txt'
url2 <- 'https://dl.dropboxusercontent.com/s/kal0605y97erj8k/billboard.txt'
url3 <- 'https://dl.dropboxusercontent.com/s/eja3qj2ewbibetd/metros.txt'
url4 <- 'https://dl.dropboxusercontent.com/s/8o4yusotr647m9f/weeks.txt'

# check file exist and if not download data
if (!file.exists('artists.txt')) download.file(url1, 'artists.txt', mode = 'wb')
if (!file.exists('billboard.txt')) download.file(url2, 'billboard.txt', mode = 'wb')
if (!file.exists('metros.txt')) download.file(url3, 'metros.txt', mode = 'wb')
if (!file.exists('weeks.txt')) download.file(url4, 'weeks.txt', mode = 'wb')

# read.table() reads a formatted text data into a data frame
artists <- read.table('artists.txt', header = T, stringsAsFactors = F, sep = ',')
billboard <- read.table('billboard.txt', header = T, stringsAsFactors = F, sep = ',')
metros <- read.table('metros.txt', header = T, stringsAsFactors = F, sep = ',')
weeks <- read.table('weeks.txt', header = T, stringsAsFactors = F, sep = ',')
```

Two side points

```r
# pre-req 1: paste0() function combines two or multiple
#    character strings into one, without space
paste0("this functi", "on will paste string", "s together")

## [1] "this function will paste strings together"

# pre-req 2: assign() function is essentially '<-'
#    but works with variable names as character data
a <- 2
a

## [1] 2

assign('a', 3)
a

## [1] 3
```

For character “artists”, the following is the code body that can be repeated

```
# for character "artists"
f <- "artists"
fc <- "0r7gcf9gxk2nm4d"

# we essentially did the following
#    note that this section is written without referring to "artists"
filename <- paste0(f, '.txt')
url <- 'https://dl.dropboxusercontent.com/s/'
if (!file.exists(filename)) download.file(paste0(url, fc, "/", filename), filename, mode = 'wb')
assign(f, read.table(filename, header = T, stringsAsFactors = F, sep = ','))

# note: minor difference is that we do not have url1, url2, etc. but instead have url.
#    in this case variable 'url' is replaced in every loop
```

The full (and simple) loop

```r
files <- c("artists", "billboard", "metros", "weeks")
file_code <- c("0r7gcf9gxk2nm4d", "kal0605y97erj8k", "eja3qj2ewbibetd", "8o4yusotr647m9f")

for (i in 1:4) {

    f <- files[i]
    fc <- file_code[i]
    filename <- paste0(f, '.txt')
    url <- 'https://dl.dropboxusercontent.com/s/'
    if (!file.exists(filename)) download.file(paste0(url, fc, "/", filename), filename, mode = 'wb')
    assign(f, read.table(filename, header = T, stringsAsFactors = F, sep = ','))

}

head(billboard)

##   artistid metroid week rank
## 1      498      65    1   87
## 2      834      65    1   27
## 3      941      65    1   76
## 4     1417      65    1   17
## 5     1518      65    1   13
## 6     2100      65    1   80
```

Example 4: restructuring non-tabular data

- Tabular data are often “well-organized” and we can usually use vectorized operators on them
  - but of course, many exceptions (e.g. example 3)
- Non-tabular data are often “irregular” and requires loops
- For example, strsplt example (will talk about strsplit more on Thursday)

```
# phone numbers
numbers <- c('585-234-5678', '424-123-3452', '810-259-1234')

# split it by '-'
split.numbers <- strsplit(numbers, "-")
split.numbers

## [[1]]
## [1] "585"  "234"  "5678"
##
## [[2]]
## [1] "424"  "123"  "3452"
##
## [[3]]
## [1] "810"  "259"  "1234"

# how would you get area code?
```

Think about how you will write the body of the loop

```
# how to get one area code?
i <- 2  # say we care about the second phone number
tentative_result <- split.numbers[[i]][1]
```

Then, close the loop

```r
# loop over all three numbers
for (i in 1:3) {
        tentative_result <- split.numbers[[i]][1]
}

# What's missing?
```

Remember to pre-assign a data structure to store your outputs

```
# pre-assign the output in a vector
area_code <- character(3)       # vector of length 3 with empty characters
# loop over area code
for (i in 1:3) {
        area_code[i] <- split.numbers[[i]][1]
}

# inspect
area_code

## [1] "585" "424" "810"
```

Nesting “for loops”: can loop over multiple indices

- For example, can use multiple indices to refer to multiple dimensions in an array

```
# initialize an array
testArray <- array(NA, c(3, 3, 3))      # array of NAs (placeholder)

# triple 'for' loop
for (i in 1:3) {
    for (j in 1:3) {
        for (k in 1:3) {
            testArray[i, j, k] <- i^2 + j^3 + k^4
        }
    }
}

# see a cross-section of this
testArray[, , 2]

##      [,1] [,2] [,3]
## [1,]   18   25   44
## [2,]   21   28   47
## [3,]   26   33   52
```

- In the above case, swapping the order of i, j and k do not matter

```
# triple 'for' loop,
#    this time different indices and different order
for (j in 1:3) {
    for (k in 1:3) {
        for (i in 1:3) {
            testArray[j, i, k] <- j^2 + i^3 + k^4
        }
    }
}

# same results
testArray[, , 2]

##      [,1] [,2] [,3]
## [1,]   18   25   44
## [2,]   21   28   47
## [3,]   26   33   52

# Note that j denotes the 'i' in previous example (and i denotes 'j')
#    also note that the loop now runs over rows, then pages, then columns
#    perfectly viable because each cell calculation is independent
#    hurts readability: much easier for array operations to stick to the row-co
```

Example 5: summary of past purchase patterns

- Calculate:
  - number of times a consumer has bought a product in the past
  - highest price at which the consumer has purchased the product
- Data and result look like

id	month	price	buy	times_bought	max_accepted_price
1	1	2	1	NA	NA
1	2	1.8	0	1	2
1	3	1.9	1	1	2
1	4	1.6	0	2	2
2	1	2	0	NA	NA
2	2	1.8	0	0	NA
2	3	1.9	1	0	NA
2	4	1.6	0	1	1.9

- How do we do this? First think about which index (indices) to loop over and which code to repeat

An algorithm

- times_bought should be the cumulative sum of buy, up to the previous month
- One version could be something like this

```
# let's say i is a consumer, m is a month
#    but we will need to tell the computer this

df$times_bought[df$id == i & df$month == m] <-
    df$times_bought[df$id == i & df$month == m - 1] +
    df$buy[df$id == i & df$month == m - 1]

# I.e., times_bought is the sum of its previous value (for the same consumer)
#    and whether the consumer bought in the previous month
```

- Not the complete loop of course:
  - need to initialize the column $times_bought
  - need to specify what i and m are

```
# initialize the variable times_bought (placeholder)
df$times_bought <- NA

# tell the computer what values i and m take
#    and execute the body of the loop
for (i in 1:2) {
    for (m in 1:4) {
        df$times_bought[df$id == i & df$month == m] <-
            df$times_bought[df$id == i & df$month == m - 1] +
            df$buy[df$id == i & df$month == m - 1]
    }
}

## Error in df$times_bought[df$id == i & df$month == m] <- df$times_bought[df$id == :
replacement has length zero

# inspect the result
df

##   id month price buy times_bought
## 1  1     1   2.0   1           NA
## 2  1     2   1.8   0           NA
## 3  1     3   1.9   1           NA
## 4  1     4   1.6   0           NA
## 5  2     1   2.0   0           NA
## 6  2     2   1.8   0           NA
## 7  2     3   1.9   1           NA
## 8  2     4   1.6   0           NA
```

- What just happened?

```
# initialize the variable times_bought to zero,
#    because we need those zeros in our algorithm
df$times_bought <- 0

# tell the computer what values i and m take
#    and execute the body of the loop
for (i in 1:2) {
    for (m in 2:4) {
        df$times_bought[df$id == i & df$month == m] <-
            df$times_bought[df$id == i & df$month == m - 1] +
            df$buy[df$id == i & df$month == m - 1]
    }
}

# and we need to replace initial value back to NA
df$times_bought[df$month == 1] <- NA

# inspect the result
df

##   id month price buy times_bought
## 1  1     1   2.0   1           NA
## 2  1     2   1.8   0            1
## 3  1     3   1.9   1            1
## 4  1     4   1.6   0            2
## 5  2     1   2.0   0           NA
## 6  2     2   1.8   0            0
## 7  2     3   1.9   1            0
## 8  2     4   1.6   0            1
```

"So," you say, "loops are straightforward to understand, then why not loop all the time?"

economics.sas.upenn.edu/~jesusfv/comparison

Table 1: Average and Relative Run Time (Seconds)

	Mac			Windows		
Language	Version/Compiler	Time	Rel. Time	Version/Compiler	Time	Rel. Time
C++	GCC-4.9.0	0.73	1.00	Visual C++ 2010	0.76	1.00
	Intel C++ 14.0.3	1.00	1.38	Intel C++ 14.0.2	0.90	1.19
	Clang 5.1	1.00	1.38	GCC-4.8.2	1.73	2.29
Fortran	GCC-4.9.0	0.76	1.05	GCC-4.8.1	1.73	2.29
	Intel Fortran 14.0.3	0.95	1.30	Intel Fortran 14.0.2	0.81	1.07
Java	JDK8u5	1.95	2.69	JDK8u5	1.59	2.10
Julia	0.2.1	1.92	2.64	0.2.1	2.04	2.70
Matlab	2014a	7.91	10.88	2014a	6.74	8.92
Python	Pypy 2.2.1	31.90	43.86	Pypy 2.2.1	34.14	45.16
	CPython 2.7.6	195.87	269.31	CPython 2.7.4	117.40	155.31
R	3.1.1, compiled	204.34	280.90	3.1.1, compiled	184.16	243.63
	3.1.1, script	345.55	475.10	3.1.1, script	371.40	491.33
Mathematica	9.0, base	588.57	809.22	9.0, base	473.34	626.19
Matlab, Mex	2014a	1.19	1.64	2014a	0.98	1.29
Rcpp	3.1.1	2.66	3.66	3.1.1	4.09	5.41
Python	Numba 0.13	1.18	1.62	Numba 0.13	1.19	1.57
	Cython	1.03	1.41	Cython	1.88	2.49
Mathematica	9.0, idiomatic	1.67	2.29	9.0, idiomatic	2.22	2.93

Speed issues: avoid loop if you can do it without

```
# let's look at a super long lifetime value
TIME <- 1E7
CLV <- numeric(TIME)    # initialize an numeric vector of zeros
CLV[1] <- exp(2)

# calculate CLV
delta <- (1 - 0.05) / (1 + 0.1) # recall discount factor
start_time <- proc.time()
for (t in 1:(TIME-1)) {
    CLV[t+1] <- CLV[t] + delta^t * exp(2 - 0.1*t)
}
proc.time() - start_time

##    user  system elapsed
##    2.05    0.00    2.04

# check answer
CLV[TIME]

## [1] 33.80953
```

Speed compared to vectorized operations

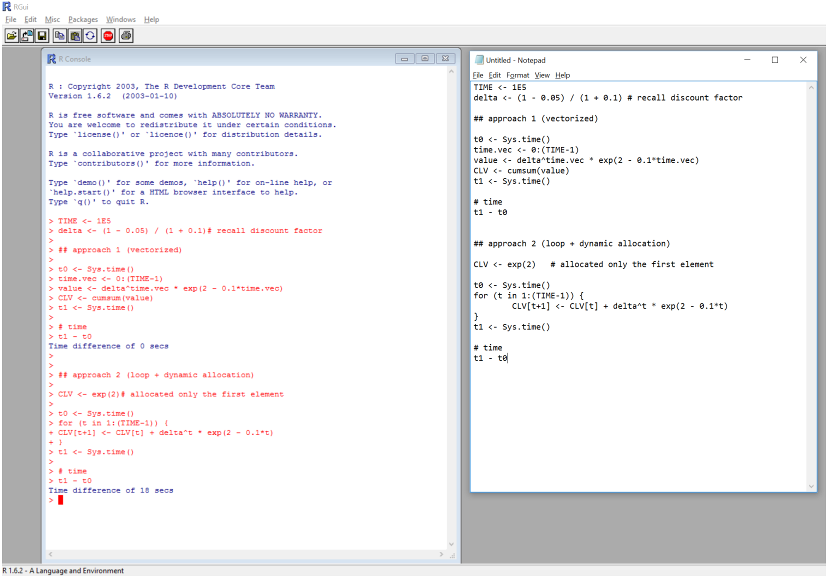
```
# compared to vectorized operations
start_time <- proc.time()
time.vec <- 0:(TIME-1)  # this is c(1, 2, ..., T)
value <- delta^time.vec * exp(2 - 0.1*time.vec) # starts from time zero
CLV <- cumsum(value)    # sum from first to current element
proc.time() - start_time

##    user  system elapsed
##    1.03    0.00    1.03

# check answer
CLV[TIME]

## [1] 33.80953
```

Loops are slow! Use vectorized operations if you can

Avoid dynamically expanding output size

```
# output not completely pre-allocated
CLV <- exp(2)   # allocated only the first element

# same for loop is much slower
start_time <- proc.time()
for (t in 1:(TIME-1)) {
    CLV[t+1] <- CLV[t] + delta^t * exp(2 - 0.1*t)
}
proc.time() - start_time

##    user  system elapsed
##    3.72    0.14    3.86

# check answer
CLV[TIME]

## [1] 33.80953
```

However, R internal has improved drastically



Can easily nest repetition and conditional statements[2]

```
# sqrt or abs
for (i in c(-9, -4, -1, 0, 1, 4, 9)) {
    if (i >= 0) {
        print(sqrt(i))
    } else {
        print(abs(i))
    }
}

## [1] 9
## [1] 4
## [1] 1
## [1] 0
## [1] 1
## [1] 2
## [1] 3
```

[2] Suppose you don't have ifelse(); now you can't vectorize this

Next, break and repeat

Next: go to the next iteration

```
# next can be used in any loops
for (iterator in times) {
    do_something1
    if (condition) next
    do_something2
}
```

For and next

```
# only compute square root for positive i's
for (i in c(-9, -4, -1, 0, 1, 4, 9)) {
    if (i < 0) next
    print(sqrt(i))
}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
```

For and break

```r
# only compute square root for negative i's
for (i in c(9, 4, 1, 0, -1, -4, -9)) {
    if (i < 0) break
    print(sqrt(i))
}
```

```
## [1] 3
## [1] 2
## [1] 1
## [1] 0
```

Repeat-break

- "Repeat" statement repeats a group of expressions infinitely
- Until a "break" statement exits the loop

```
# repeat-break loop structure
repeat {
    keep_doing_something
    if (stop_condition) break
}
```

Same old example: print and increase i as long as it's below 3

```
# initialize i
i <- 1

# repeat-break version
repeat {
    print(i)
    i <- i + 1
    if (i >= 3) break
}

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

Don't try this at home![3]

```r
# repeat must include exit condition!
i <- 1
repeat {
    print(i)
    i <- i + 1
}
```

[3] If you do and the R session won't stop running, either:
1) hit ESC in RStudio
2) open Task manager and end 'RStudio R Session' (under 'Processes')

```
## [1] 21353125
## [1] 21353126
## [1] 21353127
## [1] 21353128
## [1] 21353129
## [1] 21353130
## [1] 21353131
## [1] 21353132
## [1] 21353133
## [1] 21353134
## [1] 21353135
## [1] 21353136
## [1] 21353137
## [1] 21353138
## [1] 21353139
## [1] 21353140
## [1] 21353141
## [1] 21353142
## [1] 21353143
## [1] 21353144
## [1] 21353145
## [1] 21353146
## [1] 21353147
## [1] 21353148
## [1] 21353149
## [1] 21353150
## [1] 21353151
## [1] 21353152
## [1] 21353153
## [1] 21353154
## [1] 21353155
## [1] 21353156
## [1] 21353157
## [1] 21353158
## [1] 21353159
## [1] 21353160
## [1] 21353161
## [1] 21353162
## [1] 21353163
```

Your turn: equivalence among loop statements

While and repeat-break

```r
# repeat-break
repeat {
    keep_doing_something
    if (continue_condition == FALSE) break
}

# while: how to write this?
```

While and for

```
# while
while (iterator <= limit) {
    expression(iterator)
    iterator <- iterator_next
}

# for: how to write this?
```

Repeat and for

```
# for loop
for (iterator in 1:limit) {
    expression(iterator)
}

# repeat: how to write this?
```

Summary

- Some important repetition structures
  - for (i in vector/list) { do something }
  - while (condition is TRUE) { do something }
  - repeat { do something ; if (some exit condition) break }
- Which repetition structure to use?
  - a lot of it depends on the task you do
  - is there a clearly-defined continue condition? Use while!
  - does the body depend on a iterator? Use for!
  - repeat-break is less common

Exercise and assignment

- Example 4 (past purchase behavior) and Assignment 3
  - construct times_bought by following my code, understand what we did
  - construct max_accepted_price using a similar structure
  - next: is this the most efficient / least cumbersome structure? Try to improve on my code
  - for the last point, also see Assignment 3, where you are required to use only one index variable