PairProgamming_Loops

William Hancock, Zachary D'Urso

2024-11-5

Loops

pair programming exercise for DSE5002, Module 3

For Loops

These work when we want to repeat an operation for a known number of trials, or for each item in a list or array

for(array) { codeblock }

traditionally i,j, and k are always used as loop variables, integer counters of a loop

This loops prints the first 10 integers and their square

```
for(i in 1:10)
{
  cat(i," ",i^2,"\n")
}
```

```
## 1   1
## 2   4
## 3   9
## 4   16
## 5   25
## 6   36
## 7   49
## 8   64
## 9   81
## 10   100
```

This is a poor example though! We really shouldn't ever do this

We should be using a vector operation here to compute the squared values

Here is a much more "R" style vectorized calculation

```
x=1:10
y=data.frame(x=x,xsq=x^2)
y
```

x <int>	xsq <dbl>
1	1

x <int>	xsq <dbl>
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81
10	100

1-10 of 10 rows

We can iterate through lists

```
my_list=list("a","b",3,"c")
for(i in my_list)
{
  cat(i,"\n")
}
```

```
## a
## b
## 3
## c
```

We could put an if statement within a loop

Before running this code, predict what it will do

```
my_list=list("a","b",3,"c")
for(i in my_list)
{
  if(is.character(i))
  {
    cat(i,"\n")
  }
}
```

```
## a
## b
## c
```

Also notice how I use indentation to make the code easier to read

Each block is indented several characters, and the opening and closing curly brackets line up.

This greatly improves readability- be sure to do this.

Nested for loops

when one for loop is within another

```
for(i in seq(from=0,to=15,by=5))
{
  for(j in 1:3)
  {
    cat(j+i, "\n")
  }
}
```

```
## 1
## 2
## 3
## 6
## 7
## 8
## 11
## 12
## 13
## 16
## 17
## 18
```

Nested loops can get a bit untidy.

Jeremiah's example of a nested loop makes use of the built-in variables LETTERS (uppercase) and letters (lowercase)

```
letters[1:10]
```

```
##  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"
```

```
LETTERS[21:26]
```

```
## [1] "U" "V" "W" "X" "Y" "Z"
```

Here is Jeremiah's first example of creating combinations of the letters

```
# nested for loop (bad bad bad
# --> need to using hashing to eliminate complexity)
x4 <- character()                                          # Create empty data object
loop_work <- 0                                             # loopwork tracks how many steps
for(i in 1:5) {                                            # Head of first for-loop
  for(j in 1:5) {                                          # Head of nested for-loop
    loop_work <- loop_work + 1
    x4 <- c(x4, paste(LETTERS[i], letters[j], sep = "_"))  # Code block
  }
}
print(x4)
```

```
##  [1] "A_a" "A_b" "A_c" "A_d" "A_e" "B_a" "B_b" "B_c" "B_d" "B_e" "C_a" "C_b"
## [13] "C_c" "C_d" "C_e" "D_a" "D_b" "D_c" "D_d" "D_e" "E_a" "E_b" "E_c" "E_d"
## [25] "E_e"
```

```
print(loop_work)
```

```
## [1] 25
```

A hash table is a computer structure that allows information storage and retrieval in a small and nearly constant time interval

A “key” is used to create a “hash”, a mathematical function of the key, that is used to access or look up the “value” associated with the hash

Here is an example

```
require('hash')
```

```
## Loading required package: hash
```

```
## hash-2.2.6.3 provided by Decision Patterns
```

```
h<-hash('a'=1,'b'=2,'c'=3)
h['a']
```

```
## <hash> containing 1 key-value pair(s).
##   a : 1
```

```
require(hash)

h <- hash()
x5 <- c()
hash_work <- 0

for(i in 1:5){
  hash_work <- hash_work + 1          # count the number of assignments
  h[LETTERS[i]] <- letters[1:5]       # set letters 1-5 to be the hash contents
                                      # at the key locations of each value in
                                      # the first five values in Letters
}

#What is in H right now?
cat("After the first loop, h has ")
```

```
## After the first loop, h has
```

```
print(h)
```

```
## <hash> containing 5 key-value pair(s).
##   A : a b c d e
##   B : a b c d e
##   C : a b c d e
##   D : a b c d e
##   E : a b c d e
```

```
# how many operation used so far

cat("So far, there are ",hash_work," operations carried out\n")
```

```
## So far, there are  5  operations carried out
```

```
print("What are the names of the hash items")
```

```
## [1] "What are the names of the hash items"
```

```
names(h)
```

```
## [1] "A" "B" "C" "D" "E"
```

```
#looking at this, each entry in the hash has the five lowercase letters

#we can now go though the hash, and paste the lowercase letters onto the
# name of the hash items



for(j in 1:length(h)){
  hash_work <- hash_work + 1
  x5 <- c(x5,paste(names(h)[j],h[[LETTERS[j]]],sep='_'))
}
print(hash_work)
```

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

```
print(loop_work)
```

```
## [1] 25
```

```
# note we could have used the keys of the hash, rather than the names
# using the keys() makes more sense to me
#  x5 <- c(x5,paste(keys(h)[j],h[[LETTERS[j]]],sep='_'))

# the paste command is being used to add the contents of the hash at the key
#key value to the key value itself.   Since each key value has only one value,
# the paste function recycles the contents of names(h)[j] or keys(h)[j]
```

Hashing

What is really going on here?

Instead of nesting the two loops,

the first loop sets up the hash

a.) for each item we want in the second loop (LETTERS from 1 to 5) we use the item name from the 2nd loop as the hash key for the the vector value that will be iterated in the first loop (letters[1:5])

the second loop is not nested, it uses vector operations, extracting the second loop value from names(h) or keys(h) and the first loop values from the hashed storage

b.) Then in the second loop, loop over the items in the hash, using either names(h) or keys(h) to get the jth key, and then use a vector operation on the item stored at the key value

Benefits

1.) This should be faster 2.) While it looks unfamiliar initially, it is actually easier to read than a nested loop once you get used to the idea of a hash.

Python- has a data structure called a Dictionary that used this key-value system and hashed data look-up just as the hash() does in R

Timing

Let's check the timing on this using tic-toc

```
require(tictoc)
```

```
## Loading required package: tictoc
```

```
##
## Attaching package: 'tictoc'
```

```
## The following object is masked from 'package:hash':
##
##     clear
```

```
tic()
# nested for loop (bad bad bad
# --> need to using hashing to eliminate complexity)
x4 <- character()                                        # Create empty data object
loop_work <- 0                                           # loopwork tracks how many steps
for(i in 1:5) {                                          # Head of first for-loop
  for(j in 1:26) {                                        # Head of nested for-loop
    loop_work <- loop_work + 1
    x4 <- c(x4, paste(LETTERS[i], letters[j], sep = "_"))  # Code block
  }
}
toc()
```

```
## 0 sec elapsed
```

```
tic()
h <- hash()
x5 <- c()
hash_work <- 0

for(i in 1:5){
  hash_work <- hash_work + 1          # count the number of assignments
  h[LETTERS[i]] <- letters[1:5]       # set letters 1-5 to be the hash contents
                                      # at the key locations of each value in
                                      # the first five values in Letters
}

for(j in 1:length(h)){
  hash_work <- hash_work + 1
  x5 <- c(x5,paste(names(h)[j],h[[LETTERS[j]]],sep='_'))
}
toc()
```

```
## 0.02 sec elapsed
```

Hmm, that's interesting , no?

It does make use of vectorized operations, and the speed advantages aren't particularly noticeable on small data sets, such as we have here.

The whole idea of key-value pairs and hashed memory access is present in many NOSQL databases, so you will definitely see it again.

This discussion is interesting

https://www.r-bloggers.com/2015/03/hash-table-performance-in-r-part-i/#google_vignette (https://www.r-bloggers.com/2015/03/hash-table-performance-in-r-part-i/#google_vignette)

While Loops

Getting back to loops a bit

if we don't know how many times we need to run the loop, but we do have an ending condition we can test for, we can use a while loop

Suppose i want to compute the cube roots of the integers up until the root is 3.3 or higher.

Start with x=0, y=0

while (y<=3.3)

increase x by 1 set y to be the cube root

```
x=0
y=0

while(y<=3.3)
{
  cat(x," ",y,"\n")
  x=x+1
  y=x^(1/3)

}
```

```
## 0   0
## 1   1
## 2   1.259921
## 3   1.44225
## 4   1.587401
## 5   1.709976
## 6   1.817121
## 7   1.912931
## 8   2
## 9   2.080084
## 10   2.154435
## 11   2.22398
## 12   2.289428
## 13   2.351335
## 14   2.410142
## 15   2.466212
## 16   2.519842
## 17   2.571282
## 18   2.620741
## 19   2.668402
## 20   2.714418
## 21   2.758924
## 22   2.802039
## 23   2.843867
## 24   2.884499
## 25   2.924018
## 26   2.962496
## 27   3
## 28   3.036589
## 29   3.072317
## 30   3.107233
## 31   3.141381
## 32   3.174802
## 33   3.207534
## 34   3.239612
## 35   3.271066
```

Hhmm, I wonder if there was a way to do this in a vectorized form?

```
#use floor of 3.3^3 to find the largest integer with a cube root of less than
# 3.3

xmax=floor(3.3^3)

x=0:xmax
data.frame(x=x,y=x^(1/3))
```

x <int>	y <dbl>
0	0.000000

x <int>	y <dbl>
1	1.000000
2	1.259921
3	1.442250
4	1.587401
5	1.709976
6	1.817121
7	1.912931
8	2.000000
9	2.080084

But this is a simple example….

You can nest a for loop in a while, or a while in a for…

hash table example

Counting the number of times each word appears in a document or list

```
word_list=c("always","apple","appeared","apple","apple","apple","anxious","always","appeared","a
nd","all")

h=0
rm(h)
h<-hash()

for(word in word_list)        # loop through all the entries in the list
{
   if(!has.key(word,h))       # see if a word is in use as a key
  {
    h[[word]]<-1              # this is the first time we have seen it,
  }else
  {
    h[[word]]<-h[[word]]+1    # we have seen it before, increase the count
                              # by one for this word
  }
}

h
```

```
## <hash> containing 6 key-value pair(s).
##   all : 1
##   always : 2
##   and : 1
##   anxious : 1
##   appeared : 2
##   apple : 4
```

The hash table saves time by computing a hash of the variable word and using that to look up the current count. We don’t have to search through all the words already in use in the hash to find the count, we just look it up use the hash value.