R Training part 2

Functions, Loops, and more on Sapelo2

Goals of Training

By the end of this training you should be able to:

- Use loops and conditional logic
- Use basic vectorization
- Create your own functions
- Find and replace text
- Monitor your code
- Use what you have learned to run a batch job on Sapelo2

Control structures

- `if` and `else`: testing a condition and acting on it
- `for`: execute a loop a fixed number of times
- `while`: execute a loop *while* a condition is true
- `repeat`: execute an infinite loop (must `break` out of it to stop)
- `break`: break the execution of a loop
- `next`: skip an iteration of a loop

if(<condition>){
Do something
}

```
> x = 3
> if(x > 0){print("That's a nice positive number")}
[1] "That's a nice positive number"
>
```

Can add else Clause

```
> x = -3
> if(x > 0){print("That's a nice positive number")
+ }else{print("That's a nice negative number")}
[1] "That's a nice negative number"
>
```

ifelse()

```
> x=3
> ifelse(x>0, "positive", "negative")
[1] "positive"
>
```

Multi if statements
Or
dplyr:case_when

```
> x = 0
> if(x<0)("negative number")
> if(x>0)("positive number")
> if(x==0)("It is zero")
[1] "It is zero"
>
```

```
> case_when(
 x < 0 ~ "negative",
 x == 0 ~ "zero",
 x > 0 ~ "positive"
)
[1] "zero"
>
```

For Loops

for(x in list,vector,etc){

Do something involving x}

```
> names = c("keeko","maria","dima","ellen")
> for(name in names){
      cat("hello",name,". ")
  }
hello keeko . hello maria . hello dima . hello ellen . >
```

If statement INSIDE a loops! & is used for “and” and | is used for “or”

```
> numbers = c(-2,-1,0,1,2,3)
> for(i in numbers){
    if( (i > -1 & i < 1) | i == 3){print(i)}
  }
[1] 0
[1] 3
>
```

While Loops

```
> count = 0
> while(count < 5){
+         print(count)
+         count = count + 1 }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
```

Careful about infinite loops!

Next and Break

Next skips iterations

```
> for(num in 1:10){
        if( num < 4){next}
        print(num)
  }
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
>
```

Break exits a Loop

```
> for(num in 1:10){
        if( num > 4){break}
        print(num)
  }
[1] 1
[1] 2
[1] 3
[1] 4
>
```

Useful Built in DataSets

To see available datasets, run

```
data()
```

To load a dataset we can run

```
data(iris)
```

```
> head(iris)
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
>
```

Vectorization

Allows efficient calculations to occur.

Vectorization is often built into functions.

Much Faster Than Loops!

```
> testnumbers
 [1]  1  2  3  4  5  6  7  8  9 10
> testnumbers + 3
 [1]  4  5  6  7  8  9 10 11 12 13
```

```
> testnumbers[testnumbers<5]
[1] 1 2 3 4
>
```

```
> testnumbers < 5
 [1]  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
> abs_loop <- function(vec){
+   for (i in 1:length(vec)) {
+     if (vec[i] < 0) {
+       vec[i] <- -vec[i]
+     }
+   }
+   vec
+ }
>
>
>
> abs_sets <- function(vec){
+   negs <- vec < 0
+   vec[negs] <- vec[negs] * -1
+   vec
+ }
>
> long <- rep(c(-1, 1), 50000)
> system.time(abs_loop(long))
   user  system elapsed
  0.012   0.000   0.012
> system.time(abs_sets(long))
   user  system elapsed
  0.001   0.000   0.001
>
```

Apply Functions

Input includes the dataframe, whether the function should be applied row or column wise and the name of the function.

Faster than Loops!

```
> x = 1:10
> y = 1:10
> z = 1:10
> test = data.frame(x,y,z)
> test
    x  y  z
1   1  1  1
2   2  2  2
3   3  3  3
4   4  4  4
5   5  5  5
6   6  6  6
7   7  7  7
8   8  8  8
9   9  9  9
10 10 10 10
>
```

```
> apply(test,1,mean)
 [1]  1  2  3  4  5  6  7  8  9 10
> apply(test,2,mean)
  x   y   z
5.5 5.5 5.5
>
```

Create Functions inside apply function

```
> apply(test,2, function(x) mean(x)-1)
  x   y   z
4.5 4.5 4.5
>
```

Functions

As you saw with apply(), a function can be an argument in another function!
Functions can also be nested.
<name> = function(arguments){
    Evaluate this code
}

The return value is the last
Expression executed.

```
> my_first_function = function(x){
              cat("I love the variable",x )}
> my_first_function(3)
I love the variable 3>
> my_first_function("hello")
I love the variable hello>
> my_first_function(names)
I love the variable keeko maria dima ellen>
```

Can explicitly select return value
With return()

```
> my_second_function = function(x){
              doubled = x*2
              doubled}
> my_second_function(3)
[1] 6
>
```

Multiple arguments

```
> my_third_function = function(x,y){
                newlist = x*y
                newlist}
> my_third_function(1:3,3)
[1] 3 6 9
>
```

Default values

```
> my_fourth_function = function(x,y=3){
                newlist = x*y
                newlist}
> my_fourth_function(1:3)
[1] 3 6 9
>
```

Regular expressions

```
> grep("e",testPeople$name)
[1] 1 3 4
> grepl("e",testPeople$name)
[1]  TRUE FALSE  TRUE  TRUE
>
```

```
> sub("e","p",testPeople$name)
[1] "kpeko" "Anna"  "Ellpn" "Lukp"
> gsub("e","p",testPeople$name)
[1] "kppko" "Anna"  "Ellpn" "Lukp"
>
```

Monitoring code

The traceback() function slows roughly where an error occurred
debug() function initiates an interactive debugger:

- `n` executes the current expression and moves to the next expression
- `c` continues execution of the function and does not stop until either an error or the function exits
- `Q` quits the debugger

Use undebug() to turn off debugger
system.time() contains user time and elapsed time(time passing for user)