R Basics Part 2

Functions, Loops, and more on Sapelo2

Hello! If you have come across this document, you are likely a user of the Sapelo2 supercomputer at the University of Georgia who has already taken the R Basics Part 1 Training. This training builds on concepts covered in R Training Part1 so make sure you have completed this training before you jump into this document.
If you would like to take R Training part one, instructions to sign up are here:
https://wiki.gacrc.uga.edu/wiki/Training#How_to_Register

This training is given as a presentation. The slide number which corresponds to the section will be printed in the left margins of this document. The slides are sent out before training but can also be found here:

https://wiki.gacrc.uga.edu/images/b/ba/R_Language_Basics_PowerPoint_v1.0.pdf

(Slide 1)

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

Control structures allow you to respond to your input and execute certain pieces of code based on the input data. For example, only running a piece of code if your input meets a certain criteria or looping a piece of code over every element in your input.

Commonly used control structures are

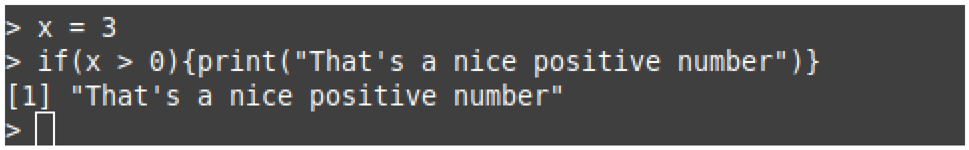
- `if` and `else`: testing a condition and acting on it
- `for`: execute a loop a fixed number of times
- `while`: execute a loop *while* a condition is true
- `repeat`: execute an infinite loop (must `break` out of it to stop)
- `break`: break the execution of a loop
- `next`: skip an iteration of a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

If Statements

The if-else structure allows you to test a condition and act on it depending on whether it's true or false. The basic structure of an if else statement is as follows:

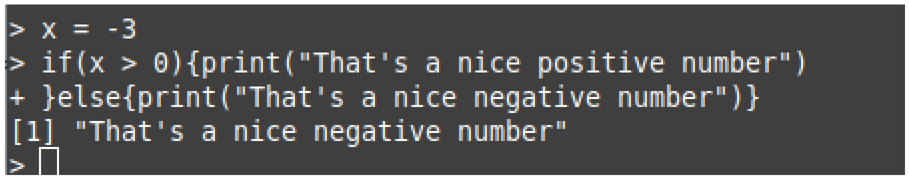
```
if(<condition>){
Do something
}
```

Where <condition> is either TRUE or FALSE. If <condition> is TRUE then the code within the curly brackets runs, and if <condition> is FALSE, the curly brackets are skipped.



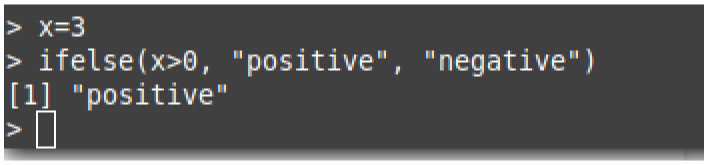
```
> x = 3
> if(x > 0){print("That's a nice positive number")}
[1] "That's a nice positive number"
> 
```

If you have code you would like to run when the condition is false then you can include an 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"
> 
```

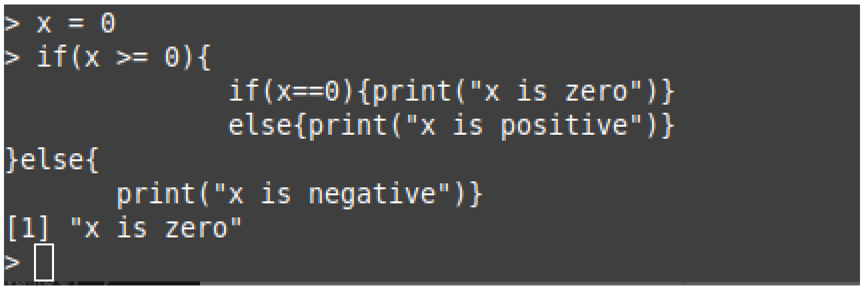
You can even combine these into an ifelse() where the first argument is the condition, the next argument is what happens if the condition is TRUE and the next for if the condition is FALSE.



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

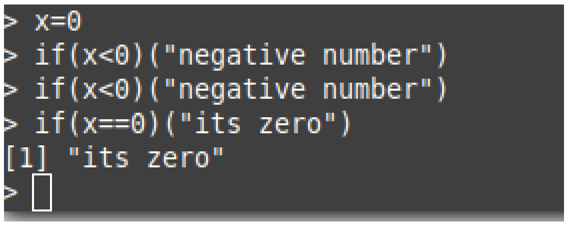
You don’t need the print() function because ifelse automatically returns the value of the TRUE argument.

But wait! What about if x = 0? We don’t want to print that x is a negative number. To fix this we can illustrate a nested if statement. This one is just to show what a nested if statement is. They can get confusing and ugly fast.



```
> x = 0
> if(x >= 0){
              if(x==0){print("x is zero")}
              else{print("x is positive")}
}else{
       print("x is negative")}
[1] "x is zero"
> 
```

It’s much better to use multiple if statements:



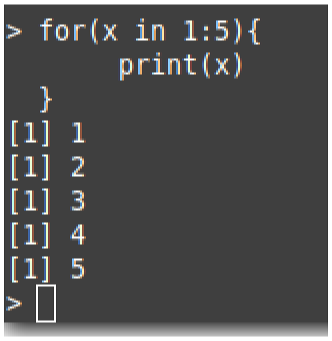
```
> x=0
> if(x<0)("negative number")
> if(x<0)("negative number")
> if(x==0)("its zero")
[1] "its zero"
> 
```

Pay close attention to the == sign. This == is different from =. A double equals sign(“equalsequals”) is used to test equivalency and will return TRUE or FALSE. The single equals sign assigns values.

Loops

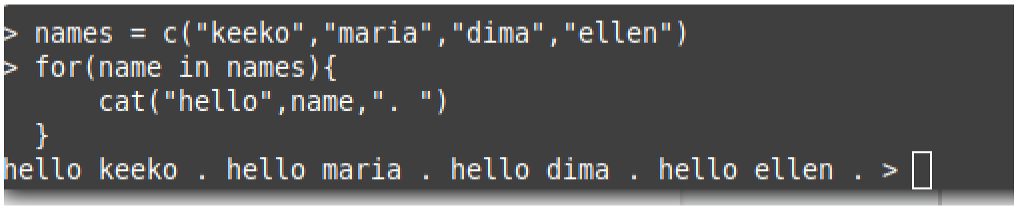
For loops are the main type of loop in R. The structure of a for loop is like this:
for(x in list,vector,etc){
Do something involving x}

The for loop sets x equal to the first value in a list or vector and then runs the code in the curly brackets. Then the loop sets the value of x to the next value in the list/vector and runs the code in the curly brackets again. This process is finished after the curly bracket code has been run for the last x in the list/vector.



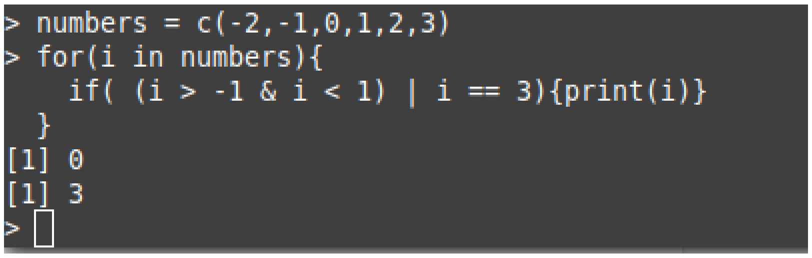
```
> for(x in 1:5){
       print(x)
  }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
> 
```

You don’t have to use x specifically, it is just a placeholder.



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

The symbol & is used for “and” and | is used for “or”
Combine these for logic fun.

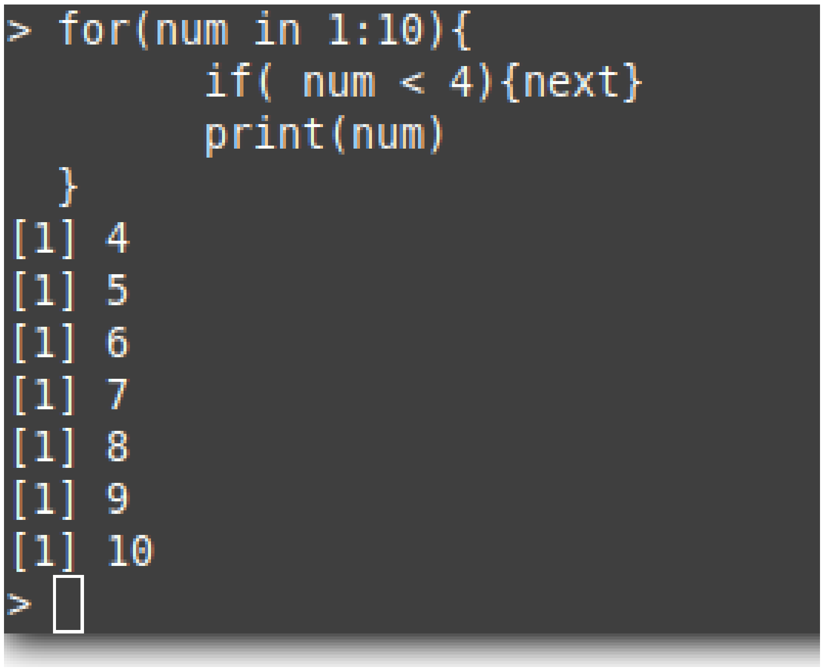


```
> numbers = c(-2,-1,0,1,2,3)
> for(i in numbers){
    if( (i > -1 & i < 1) | i == 3){print(i)}
  }
[1] 0
[1] 3
> 
```

While loops begin by testing a condition and if it is true they execute the code in the curly brackets. Then the condition is tested again and if it is again true then the code in the curly brackets runs again. This goes on until the condition is FALSE.

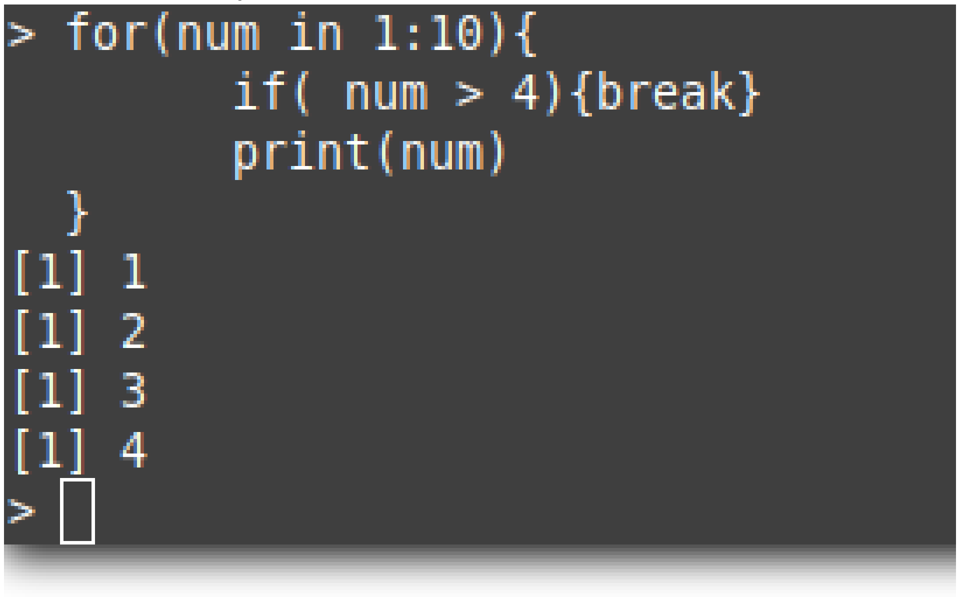
Be careful because while loops can end up in an infinite loop!

Next is used to skip certain 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
> 
```

An Example on Sapelo

The Data we will be using for this tutorial is the Iris flower data set. Not only because I love flowers but because this data set is one of the best known databases for data analysis. I'm sure many of you have seen it before and will see it again in the future, especially if you pursue machine learning. Information about the Iris data set can be found here https://archive.ics.uci.edu/ml/datasets/iris

This dataset, along with many other useful practice data, come with the R package called datasets. This package mostly comes built into your R installation. To see available datasets, run

```
data()
```

To load a dataset we can run

```
data(iris)
```

Now let's get back to the coding.
The data for this example can be found an Sapelo at /usr/local/training/R/R_part2
We want to write a script which calculates the average sepal with per species. Each Species is represented by a different csv file in our directory. We would like to iterate one piece of code over each file. Remember that

```
list.files()
```

Gives us a list of files in our directory. We can use the pattern argument to only choose files which end in .csv.

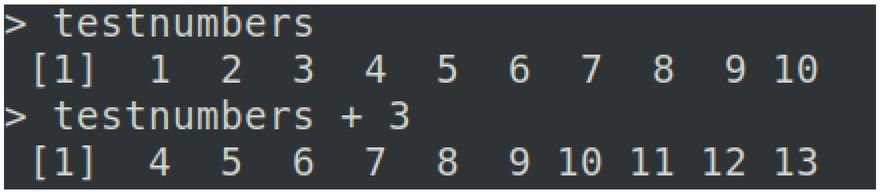
```
for(file in list.files(pattern = ".csv")){
species = read.csv(file)
print(mean(species$Sepal.Width))
}
```

Vectorization and Subsetting

(slide 20)

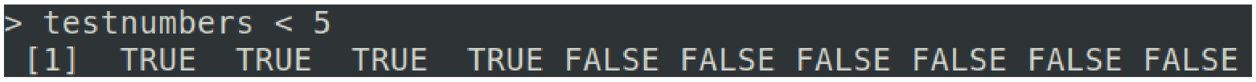
Vectorization and subsetting are features of R that allow efficient calculations to occur. Vectorization is often built into functions.

Instead of looping an action is done to each member of a vector in one command



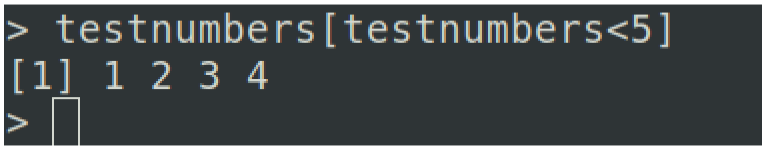
```
> testnumbers
 [1]  1  2  3  4  5  6  7  8  9 10
> testnumbers + 3
 [1]  4  5  6  7  8  9 10 11 12 13
```

Comparisons using vectors are also very fast



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

As well as subsetting a vector

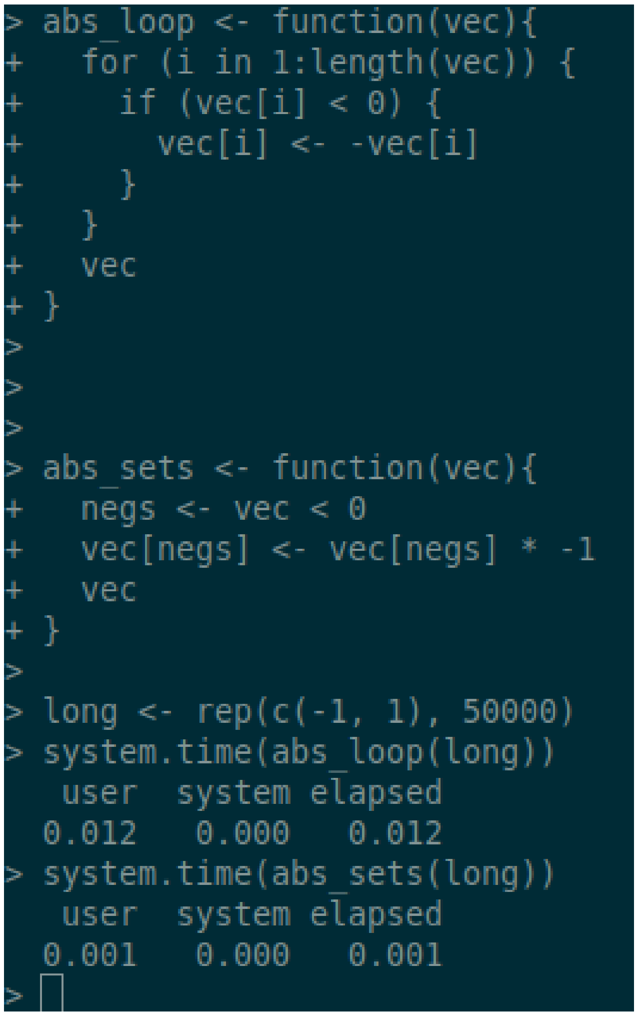


```
> testnumbers[testnumbers<5]
[1] 1 2 3 4
> 
```

Use vectors and subsetting whenever you can to speed up processes.

The following example shows two functions which take the absolute value of each number in a vector. The first, abs_loop, does not use vectorization or subsetting. The second, abs_stes does.
rep(c(-1,1),50000) creates a vector of length 50000 with values of either -1 or 1.

The function system.time() shows us that the vectorized version is twelve times as fast!



```
> 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
> 
```

So when would you use loops?

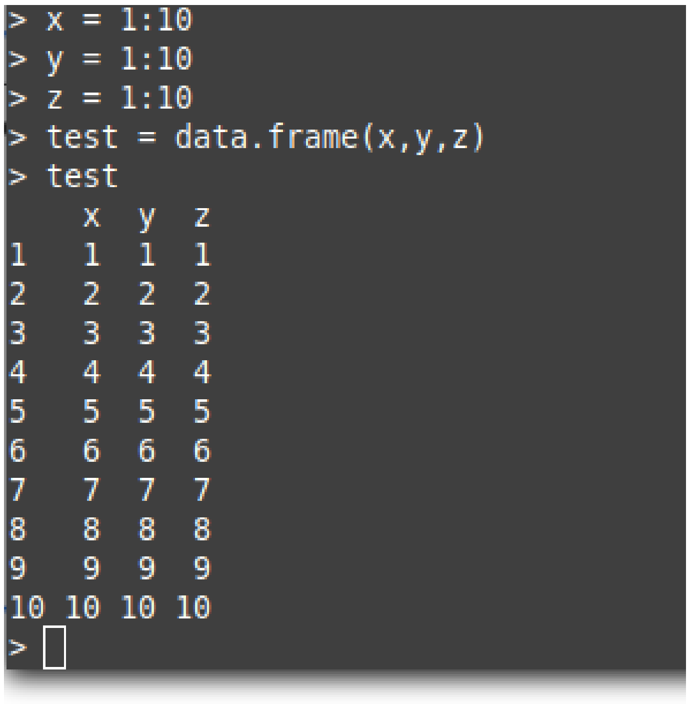
Well, firstly, you will see them a lot so it's good to recognize them. Secondly, there are plenty of examples where a loop would be better. For example, if the code you are running changes depending on results from the last iteration. You couldn't run these "in-parallel" because you need the results of an earlier iteration.

Vectorization isn't actually running in parallel, as in parallel processing. There is still a for loop going on at a deeper level(usually calling FORTRAN or C) but the deeper stuff is happening much faster than an R for loop.

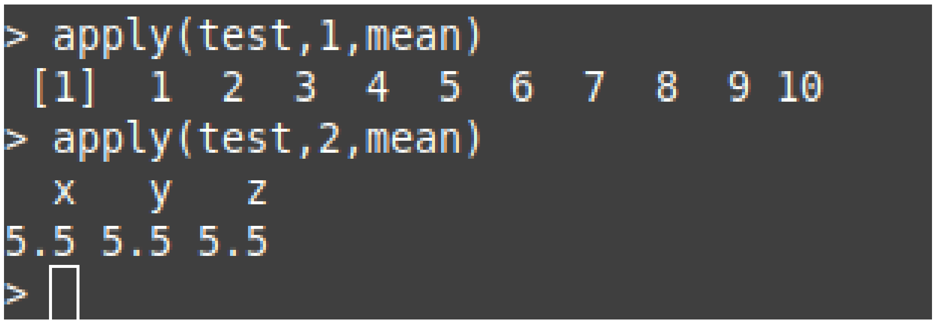
The Apply Functions

If we want to apply a function to every row or column we can use the apply() function. The input to the apply function includes the dataframe, whether the function should be applied row or column wise and the name of the function.

Apply functions are usually much faster than loops.

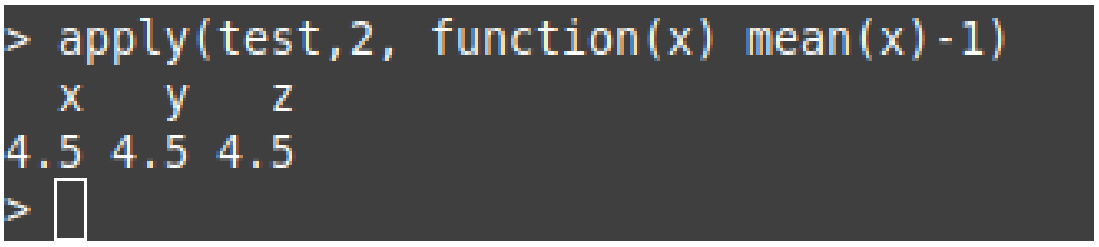


```
> x = 1:10
> y = 1:10
> z = 1:10
> test = data.frame(x,y,z)
> test
    x  y  z
1   1  1  1
2   2  2  2
3   3  3  3
4   4  4  4
5   5  5  5
6   6  6  6
7   7  7  7
8   8  8  8
9   9  9  9
10 10 10 10
> 
```



```
> 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()



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

There are many different types of apply function, also called the apply family of functions in R. Apply functions are about as fast as loops but are much cleaner. They can boost speed depending on what sort of calculations you are doing and which apply function you are using. Let's edit our SepalWidth.R code to test out the apply function:

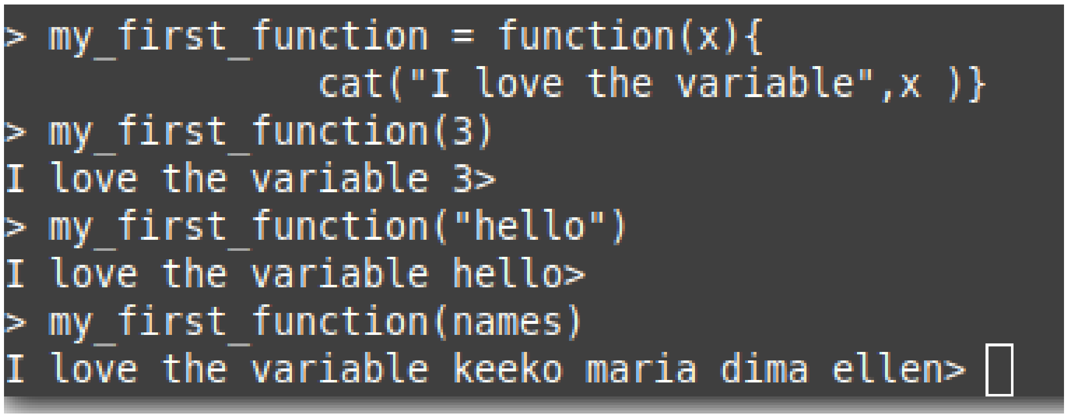
```
system.time(
for(file in list.files(pattern = ".csv")){
species = read.csv(file)
print(mean(species$Sepal.Width))
}
)

system.time(sapply(list.files(pattern = ".csv"), function(x)
{species=read.csv(x);print(mean(species$Sepal.Width))}) )
```

Functions

As you saw with apply(), a function can be an argument in another function!
Functions can also be nested, meaning we have functions that call other functions. Besides the functions which are built into R or come with packages, you can also write your own functions. The typical structure of a function is as follows:

```
<name> = function(arguements){
        Evaluate this code
}
```

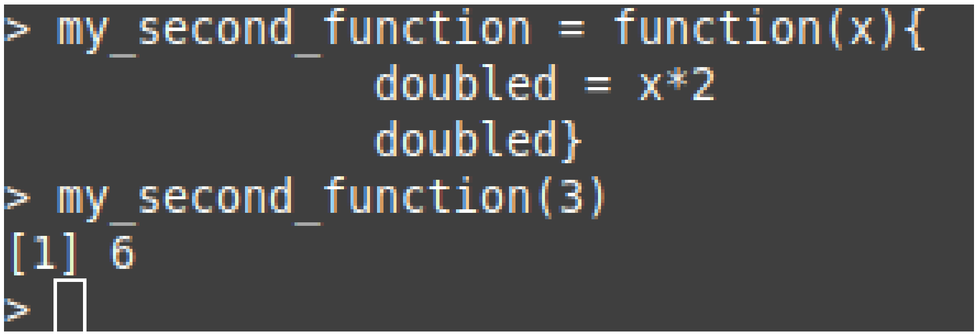


```
> my_first_function = function(x){
              cat("I love the variable",x )}
> my_first_function(3)
I love the variable 3>
> my_first_function("hello")
I love the variable hello>
> my_first_function(names)
I love the variable keeko maria dima ellen>
```

The function cat stands for concatenate and this function links (things) together in a chain or series. So Cat links things together and prints them. So in this case cat will print “I love the variable " and then print the variable.
You can also add a separator as an argument to the cat function.

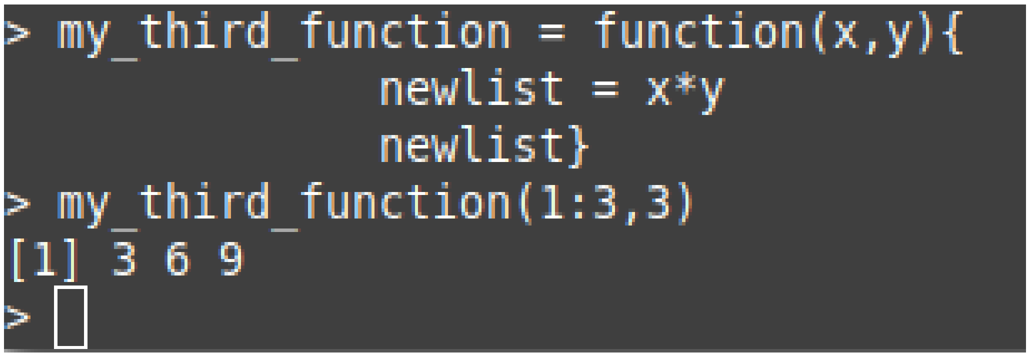
```
cat("I love the variable", names,sep = " and ")
```

Another word for your output is your return value. What your function returns to the user. In this case our function is returning the concatenated sentence.
Your function automatically returns the last expression executed. You can also explicitly set the return value with return()
Variables which are created inside functions only exist inside the function. Once the function has completed running, the variable is gone.



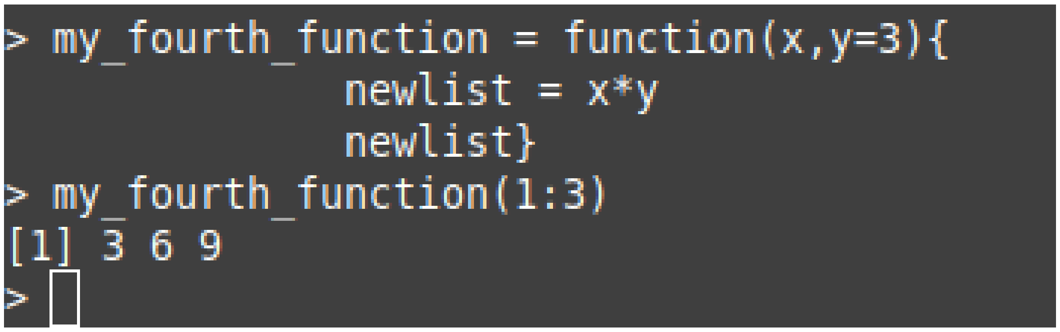
```
> my_second_function = function(x){
              doubled = x*2
              doubled}
> my_second_function(3)
[1] 6
>
```

You can have multiple inputs:



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

And Default values:



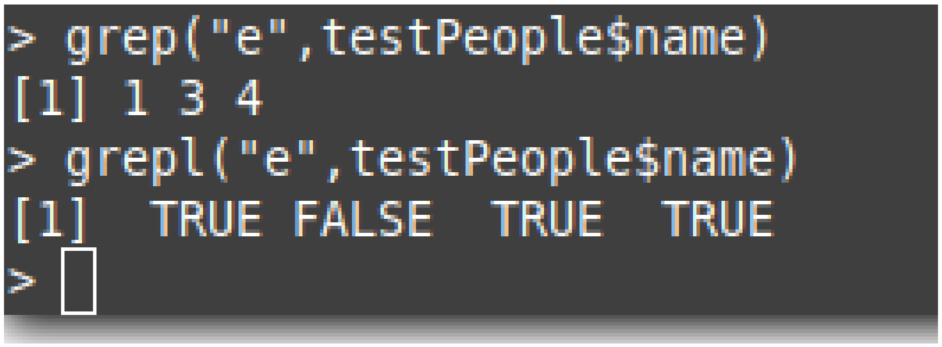
```
> my_fourth_function = function(x,y=3){
                newlist = x*y
                newlist}
> my_fourth_function(1:3)
[1] 3 6 9
> 
```

Let’s edit out SepalWidth.R function again:

```
#forloop
system.time(
for(file in list.files(pattern = ".csv")){
species = read.csv(file)
print(mean(species$Sepal.Width))
}
)
#apply method
system.time(
   sapply(list.files(pattern = ".csv"), function(x)
{species=read.csv(x);print(mean(species$Sepal.Width))})
)
#function method
Sepalmean = function(x){
data = read.csv(x)
sepalmean = mean(data$Sepal.Width)
print(sepalmean)}
system.time(
sapply(list.files(pattern="csv"),Sepalmean))
```

Regular Expressions

Regular expressions are expressions which are used for pattern matching.
One example you may have seen before in Linux is grep, which is also an R function.
Grep searches for matches within each element of a vector.



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

As you can see grep prints the position of the values that have a match.
If we want to select these values we would want something like:

```
names[c(1,3)]
```

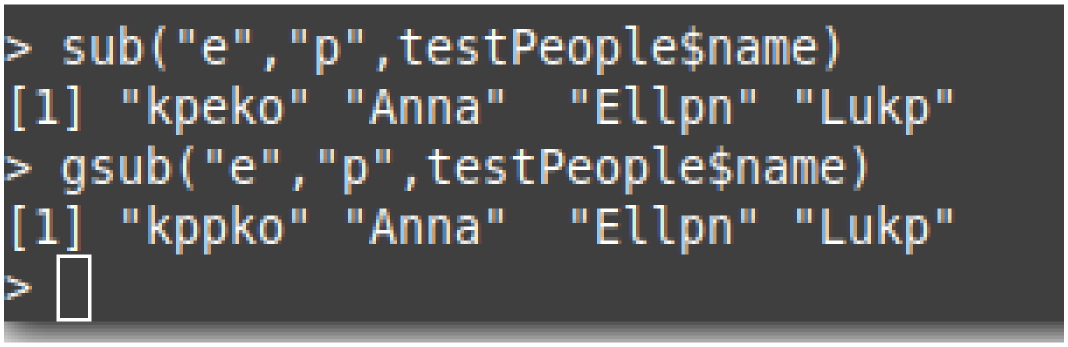
or

```
names[grep( "e" , names )]
```

or

```
grep(pattern = "Human", x =
readLines("results1.out"),value = TRUE )
```

Use sub() and gsub() to replace patterns of characters with others.



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

Let's say you have a large output and you only want to select a certain piece of it to move onto the next step in your pipeline.
For this example we will use the output of a software called Blast. This software searched through databases and found matches to a nucleotide sequence. The output is the file results1.out.
This file contains possible matches along with a score of how much each matches my input data. Let's say I know my gene is from a human so I only want to see output which has the word "human" in it.
What I would do is first use readLines instead of read csv. readLines() makes a vector where each value is a line from your file. Then I'd use grep.

```
grep(pattern = "Human", x = readLines("results1.out"),
value = TRUE )
```

These aren't great examples of regular expressions because they are very simple. There are all sorts of ways to get really specific with your searches such as searching for a word appearing only once or only the last 3 instances of a pattern.

Monitoring code

Usually functions call other functions, which call other functions. It can often get confusing where a problem stems from when there are so many layers. To show examples of monitoring code I first create a complicated function.

```
 f = function(a) g(a)
 g = function(b) h(b)
 h = function(c) i(c)
 i = function(d) {
```

```
if(!is.numeric(d)){
      stop("d must be numeric", call.= FALSE)
 }
d + 10   }
```

Running f(10) works fine but running f("h") gives an error. We can use traceback to see where the error occurred. You read the traceback() output from bottom to top.

If you would like to be able to interact with variables which you are running each layer of a function you can use a debugger. This is good for monitoring how a variable changed throughout a function. To run a function in debug mode you use the function debug().
Once a function is running in debug mode you can interact with your environment as well as type n,c, or q :

- Next, `n`: executes the next step in the function. If you have a variable named `n`, you'll need `print(n)` to display its value.

- Continue, `c`: leaves interactive debugging and continues regular execution of the function. This is useful if you've fixed the bad state and want to check that the function proceeds correctly.
- Stop, `Q`: stops debugging, terminates the function, and returns to the global workspace. Use this once you've figured out where the problem is, and you're ready to fix it and reload the code.

You can time a piece of code with system.time()
User time is how many seconds the computer spent doing your calculations. System time is how much time the operating system spent responding to your program's requests. Elapsed time is the sum of those two, plus whatever "waiting around" your program and/or the OS had to do

- If **elapsed time > user time**, this means that the CPU is waiting around for some other operations (may be external) to be done.
- If **elapsed time < user time**, this means that your machine has multiple cores and is able to use them