
R programming techniques

4

Basic R 'Built-in' functions for working with objects

- R has many built-in functions for doing simple calculations on objects. Start with a random sample of 15 numbers from 1 to 100 and try the functions below.

```
> x<-sample(100,15)
```

- Arithmetic with vectors
 - Min / Max value number in a series

```
min(x) ; max(x)
```

- Sum of values in a series

```
sum(x)
```

- Average estimates (mean / median)

```
mean(x) ; median(x)
```

- Range of values in a series

```
range(x)
```

- Variance

```
var(x)
```

- Arithmetic with vectors
 - Rank ordering

```
rank(x)
```

- Quantiles

```
quantile(x) ; boxplot(x)
```

- Square Root

```
sqrt(x)
```

- Standard deviation

```
sd(x)
```

- Trigonometry functions

```
tan(x) ; cos(x) ; sin(x)
```

Basic R 'Built-in' functions for working with data frames

- We have seen before how we can get the **names** of our variables, but for dataframes and matrices we can also get these names with **colnames**, and the names of the rows with **rownames**:

```
> names(patients)
[1] "First_Name" "Second_Name" "Full_Name"   "Sex"         "Age"         "Weight"      "Consent"
> colnames(patients)
[1] "First_Name" "Second_Name" "Full_Name"   "Sex"         "Age"         "Weight"      "Consent"
> rownames(patients)
[1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10"
```

We can get the numbers of rows or columns with **nrow** and **ncol**:

```
> nrow(patients)
[1] 10
> ncol(patients)
[1] 7
```

We can also find the length of a vector or a list with **length**, although this may give confusing results for some structures, like data frames:

```
> length(c(1,2,3,4,5))
[1] 5
> length(patients)
[1] 7
> length(patients$Age)
[1] 10
```

Remember, a data frame is a list of variables, so its length is the number of variables. The length of one of the variable vectors (like Age) is the number of observations.

Basic R 'Built-in' functions for working with data frames

We can add rows or columns to a data frame using **rbind** and **cbind**:

```
> newpatient<-c("Kate","Lawson","Kate Lawson","Female","35","62.5","TRUE")
```

```
> rbind(patients,newpatient)
```

	First_Name	Second_Name	Full_Name	Sex	Age	Weight	Consent
1	Adam	Jones	Adam Jones	Male	50	70.8	TRUE
2	Eve	Parker	Eve Parker	Female	21	67.9	TRUE
3	John	Evans	John Evans	Male	35	75.3	FALSE
4	Mary	Davis	Mary Davis	Female	45	61.9	TRUE
5	Peter	Baker	Peter Baker	Male	28	72.4	FALSE
6	Paul	Daniels	Paul Daniels	Male	31	69.9	FALSE
7	Joanna	Edwards	Joanna Edwards	Female	42	63.5	FALSE
8	Matthew	Smith	Matthew Smith	Male	33	71.5	TRUE
9	David	Roberts	David Roberts	Male	57	73.2	FALSE
10	Sally	Wilson	Sally Wilson	Female	62	64.8	TRUE
11	Kate	Lawson	Kate Lawson	Female	35	62.5	TRUE

```
> cbind(patients,10:1)
```

- We can also remove rows and columns:

```
df[-1,] # remove first row
```

```
df[, -1] # remove first column
```

Basic R 'Built-in' functions for working with data frames

Sorting a vector with **sort**:

```
> sort(patients$Second_Name)
```

```
[1] "Baker"    "Daniels"  "Davis"    "Edwards"  "Evans"    "Jones"    "Parker"    "Roberts"  "Smith"    "Wilson"
```

Sorting a data frame by one variable with **order**:

```
> order(patients$Second_Name)
```

```
[1]  5  6  4  7  3  1  2  9  8 10
```

```
> patients[order(patients$Second_Name),]
```

	First_Name	Second_Name	Full_Name	Sex	Age	Weight	Consent
5	Peter	Baker	Peter Baker	Male	28	72.4	FALSE
6	Paul	Daniels	Paul Daniels	Male	31	69.9	FALSE
4	Mary	Davis	Mary Davis	Female	45	61.9	TRUE
7	Joanna	Edwards	Joanna Edwards	Female	42	63.5	FALSE
3	John	Evans	John Evans	Male	35	75.3	FALSE
1	Adam	Jones	Adam Jones	Male	50	70.8	TRUE
2	Eve	Parker	Eve Parker	Female	21	67.9	TRUE
9	David	Roberts	David Roberts	Male	57	73.2	FALSE
8	Matthew	Smith	Matthew Smith	Male	33	71.5	TRUE
10	Sally	Wilson	Sally Wilson	Female	62	64.8	TRUE

The R workspace

- The objects we have been making are created in the R workspace.
- When we load a package, we are loading that package's functions and data sets into our workspace.
- You can see what is in your workspace with **ls**:

```
> ls()
```

- You can remove objects from the workspace with **rm**:

```
> x<-1:5
```

```
[1] 1 2 3 4 5
```

```
> rm(x)
```

```
> x
```

```
Error: object 'x' not found
```

- You can remove everything by giving **rm** a list of all the objects returned by **ls**:

```
> rm(list=ls())
```

Introducing loops

- Many programming languages have ways of doing the same thing many times, perhaps changing some variable each time. This is called **looping**.
- Loops are not used in R so often, because we can usually achieve the same thing using vector calculations.
- For example, to add two vectors together, we do not need to add each pair of elements one by one, we can just add the vectors.
- But there are some situations where R functions can not take vectors as input. For example, **read.csv** will only load one file at a time.
- What if we had ten files to load in, all ending in the same extension (like `.csv`)?

Introducing loops

- We could do this:

```
> colony<-data.frame()      # Start with empty data frame

> colony1<-read.csv("11_CFA_Run1Counts.csv")
> colony2<-read.csv("11_CFA_Run2Counts.csv")
> colony3<-read.csv("11_CFA_Run3Counts.csv")
...
> colony10<-read.csv("11_CFA_Run10Counts.csv")

> colony<-rbind(colony1,colony2,colony3,...,colony10)
```

But this will be boring to type, difficult to change, and prone to error.

- As we are doing the same thing 10 times, but with a different file name each time, we can use a **loop** instead.

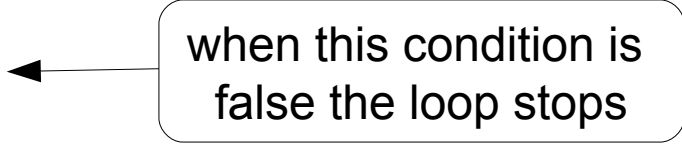
LOOPS

Commands & flow control

- R has two basic types of loop:
 - for** loop: run some code on every value in a vector
 - while** loop: run some code while some condition is true
- Here are two simple examples of these loops:

```
for (f in 1:10) {  
  print(f)  
}
```

```
f <- 1  
while ( f <= 10 ) {  
  print(f)  
  f <- f + 1  
}
```



when this condition is
false the loop stops

LOOPS

Commands & flow control

- Here's how we might use a **for** loop to load in our CSV files.
- We can look up files containing a particular substring in their name using the **dir** function:

```
dir(pattern="Counts.csv")
```

```
[1] "11_CFA_Run1Counts.csv" "11_CFA_Run2Counts.csv"  
    "11_CFA_Run3Counts.csv"
```

- So we can load all the files using a **for** loop as follows:

```
colony<-data.frame()
```

```
countfiles<-dir(pattern="Counts.csv")
```

```
for (file in countfiles) {
```

```
  t<-read.csv(file)
```

```
  colony<-rbind(colony,t)
```

```
}
```

- Here, we use a temporary variable **t** to store the data in each file, and then add that data to the main **colony** data frame.

Conditional branching

Commands & flow control

- Use an **if** statement for any kind of condition testing.
- Different outcomes can be selected based on a condition within brackets.

```
if (condition) {  
... do this ...  
} else {  
... do something else ...  
}
```

- **condition** is any logical value, and can contain multiple conditions
 - e.g. **(a==2 & b <5)**, this is a compound conditional argument

Conditional branching

Commands & flow control

For example, if we were writing a script to load an unknown set of files, using the **for** loop we wrote before, we might want to warn the user if we can't find any files with the pattern we are searching for.

Here's how we can use an **if** statement to test for this:

```
colony<-data.frame()  
countfiles<-dir(pattern="Counts.csv")  
  
if (length(countfiles) == 0) {  
  stop("No Counts.csv files found!")  
} else {  
  for (file in countfiles) {  
    t<-read.csv(file)  
    colony<-rbind(colony,t)  
  }  
}
```

The **stop** function outputs the error message and quits.

Code formatting avoids bugs!

- Code formatting is crucial for readability of loops

```
f<-26
while(f!=0){
print(letters[f])
f<-f-1 }
```

BAD!!!

```
f <- 26
while( f != 0 ){
    print(letters[f])
    f <- f-1
}
```

GOOD!

- The code between brackets {} **always** is indented, this clearly separates what is executed once, and what is run multiple times
- Trailing bracket } always alone on the line at the same indentation level as the initial bracket {
- Use white spaces to divide the horizontal space between units of your code, e.g. around assignments, comparisons

Exercise

1. Output the patients data frame, with the patients sorted in order of age, oldest first. (You may need the **rev** function.)
2. Load in the **colony** data frame using a for loop. Three of the data files are in the *Day_1_scripts* folder. Load all three files into **colony** using the for loop in the slides.
3. How many observations do you have in the **colony** data frame? Find out by counting the number of rows in **colony** using the **nrow** function.
4. You have calculated that you will have sufficient power for your analysis if you have at least 70 observations. Modify your **for** loop so it will only load files if colony has less than 70 observations in it.

Answers to exercise

1. To order the patients by decreasing age:

```
patients[rev(order(patients$Age)),]
```

3. To stop loading files after at least 70 observations have been found, use the code from the first **for** loop slide with a new **if** statement:

```
colony<-data.frame()  
countfiles<-dir(pattern="Counts.csv")  
for (file in countfiles) {  
  if ( nrow(colony) <= 70 ) {  
    t<-read.csv(file)  
    colony<-rbind(colony,t)  
  }  
}
```