Computing with Vectors_Intro_R_2_2

HDS, Mike Kozlowski

June 23, 2018

Computing with Vectors in R,

Follwoing van der Loo and de Jonge, section 2.2.1 and onward

We will look at a series of operations with R vectors, in three general categories

-mathematical operations on vectors

-simple functions on or over vectors

-subsetting, or slicing, or indexing of vectors to select specific elements of a vector These operations are important basic tools in R that will help greatly in dealing with larger data
sets since in R most data structures are combinations of vectors.

###Basic Mathematical Operations on Vectors

Most operations are carried out in an element-wise fashion

```
x=c(1,2,3,4,5,6)
y=c(0,1,0,1,0,1)
z=x+y
z
```

```
## [1] 1 3 3 5 5 7
```

Multiplication is also an element-wise operation

```
z=x*y
z
```

```
## [1] 0 2 0 4 0 6
```

So this multiplication of vectors in R is not a dot product (aka inner product, or scalar product) or a vector cross product. The dot product is available using the %*% operator

```
x%*%y
```

```
##      [,1]
## [1,]   12
```

The dot product is a major mathematical idea in linear algebra (aka Matrice Algebra) Look it up in Wikipedia or elsewhere on line.

X dot y = magnitude of x times magnitude of y times the cosine of the angle between them

Question/Action

Use the dot product to find the magnitude of x

```
mx = ( x %*% x ) ^ 0.5
mx
```

```
##          [,1]
## [1,] 9.539392
```

Find the magnitude of y

```
my = ( y %*% y ) ^ 0.5
my
```

```
##          [,1]
## [1,] 1.732051
```

Find the cosine of the angle between x and y

```
cosxy = ( x %*% y ) / ( mx * my )
cosxy
```

```
##          [,1]
## [1,] 0.726273
```

Find the angle between x and y

```
acos(cosxy)
```

```
##           [,1]
## [1,] 0.7579118
```

The vector cross-product can be computed as well, it is in the pracma package.

Leaving that aside for a moment, we can look at the addition of vectors and scalars, adding a scalar to a vector increases each element by the value of the scalar

```
w=x+3
w
```

```
## [1] 4 5 6 7 8 9
```

R does something a bit odd when we add vectors of unequal length- the smaller vector is recycled or reused in

the calculation- you have to be a bit careful about this, it can produce some unexpected results if you thought you were adding equal length vectors to one another

```
q=c(0,10,20)
w=x+q
w
```

```
## [1]  1 12 23  4 15 26
```

Notice that there is no warning of the recycling, adding two vectors of lengths differing by one can surprise you, but at least there is a warning

```
q=c(0,10,20,30,40)
w=x+q
```

```
## Warning in x + q: longer object length is not a multiple of shorter object
## length
```

```
w
```

```
## [1]  1 12 23 34 45  6
```

We can carry out some typical mathematical functions on a vector, which act in an element-wise fashion

```
log10(x)
```

```
## [1] 0.0000000 0.3010300 0.4771213 0.6020600 0.6989700 0.7781513
```

```
tan(pi/180*x)
```

```
## [1] 0.01745506 0.03492077 0.05240778 0.06992681 0.08748866 0.10510424
```

We can do logical operations the same way

```
w=3*y
w>=x
```

```
## [1] FALSE  TRUE FALSE FALSE FALSE FALSE
```

Which returns an binary array of the truth values of the operation

###Operations which summarize or describe all elements in the vector

Perhaps the most obvious is the sum function

```
x
```

```
## [1] 1 2 3 4 5 6
```

```
sum(x)
```

```
## [1] 21
```

There is also a product

```
prod(x)
```

```
## [1] 720
```

And some basic statistical functions

```
mean(x)
```

```
## [1] 3.5
```

```
median(x)
```

```
## [1] 3.5
```

Standard Deviation

```
sd(x)
```

```
## [1] 1.870829
```

Variance

```
var(x)
```

```
## [1] 3.5
```

```
summary(x)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.00    2.25    3.50    3.50    4.75    6.00
```

Stop for a moment and think about what you could do with these types of simple vector operations..

##Okay, here are a couple of quick application

####Application 1- The Z transformation

We can carry out what is called a z-transformation of the data, which rescales it to a mean of zero and as standard deviation of 1-

First we can *center* the data which means subtract the mean of the data from each element in the data, so the the mean of the centered version of the data is zero, and each element now is the *deviance* from the mean

```
xCentered=x-mean(x)
summary(xCentered)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   -2.50   -1.25    0.00    0.00    1.25    2.50
```

Now we can complete the Z-transformation by dividing xCentered by it's own standard deviation

```
xZtrans=xCentered/sd(xCentered)
summary(xZtrans)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -1.3363 -0.6682  0.0000  0.0000  0.6682  1.3363
```

```
sd(xZtrans)
```

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

####Example 2- Building a simulated data set

Suppose we want to model the flight of a ball thrown upward at an initial velocity of 10 m/s. The starting height from the ground when the ball is released is 1.6m, about shoulder height. Gravity pulls the ball down at an acceleration of -9.8 m/s^2

If h=height, and t= time

h=1/2at^2+v0t+ h0

a=-9.8, v0=10, h0=1.6

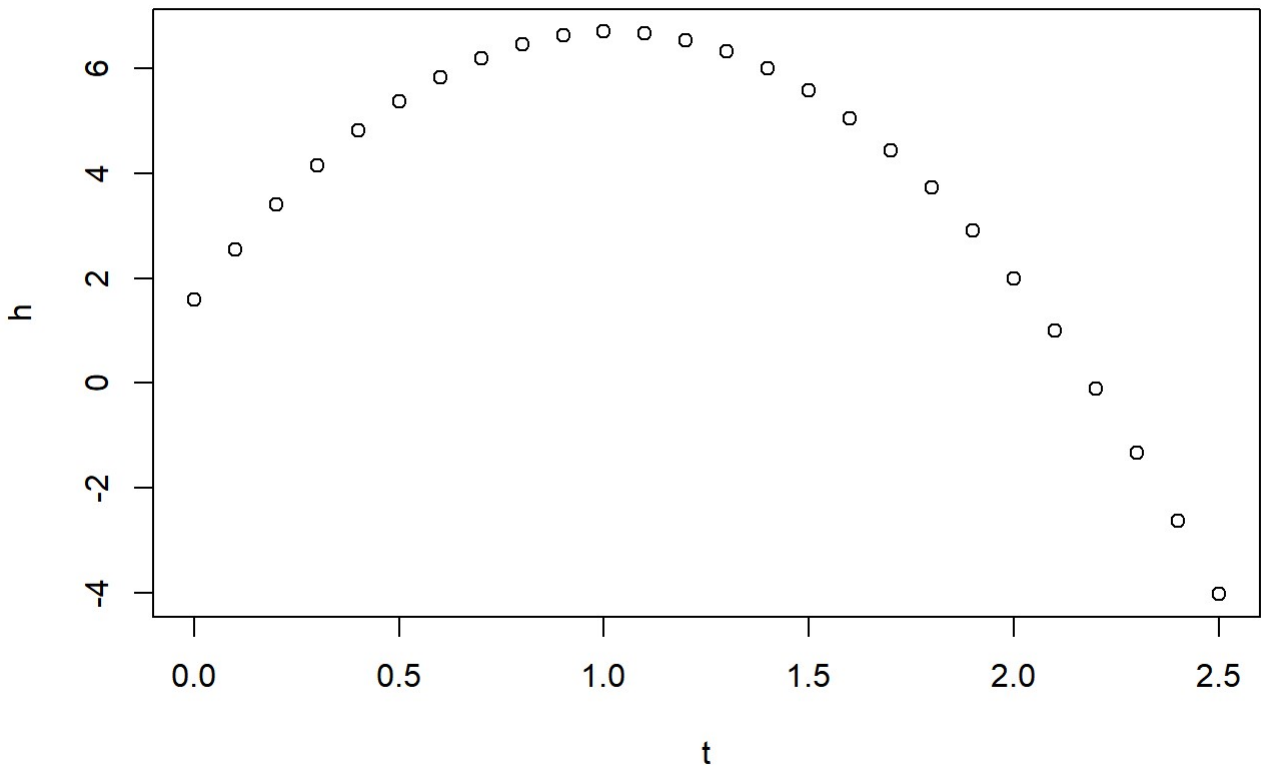
h= 1/2 (-9.8)t^2+ 10(t)

let's set up a set of t values from 0 to maybe 2.5 seconds, it should reach zero by time t=2.5 seconds at the latest. We'll take t in steps of 0.1 seconds, using seq to generate this set of times as a vector

```
t=seq(from=0,to=2.5,by=0.1)
g=-9.8
h0=1.6
v0=10
h=0.5*g*t^2+v0*t+h0
h
```

```
##  [1]  1.600  2.551  3.404  4.159  4.816  5.375  5.836  6.199  6.464  6.631
## [11]  6.700  6.671  6.544  6.319  5.996  5.575  5.056  4.439  3.724  2.911
## [21]  2.000  0.991 -0.116 -1.321 -2.624 -4.025
```

```
plot(t,h,type='p')
```



I just used the base R plot() function- ggplot2 offers a wider range of much better plots, but plot does the job here. I refer to very simple plot one uses as part of the analysis plots as “scratch” graphics- domain specialists need high quality plots, analysts often use these quick and dirty “scratch” plots- good enough for a quick look at results

But if you are really good at ggplot2 or other graphics, use them if they work for you.

Okay, so this is plot of predicted flight, but these are the “true” values of t and h, and we know there are errors in the measurements.

Let’s pretend that values of t are pretty accurate, maybe we used a computer timer, or took the height measurements from a video with a “constant” frame rate (hmm, how constant are video frame rates, really?)

Variations in timing aside, let’s suppose we did a *repeated measures study* and found that when we measured a given height over and over that the standard deviation was 0.15 meters.

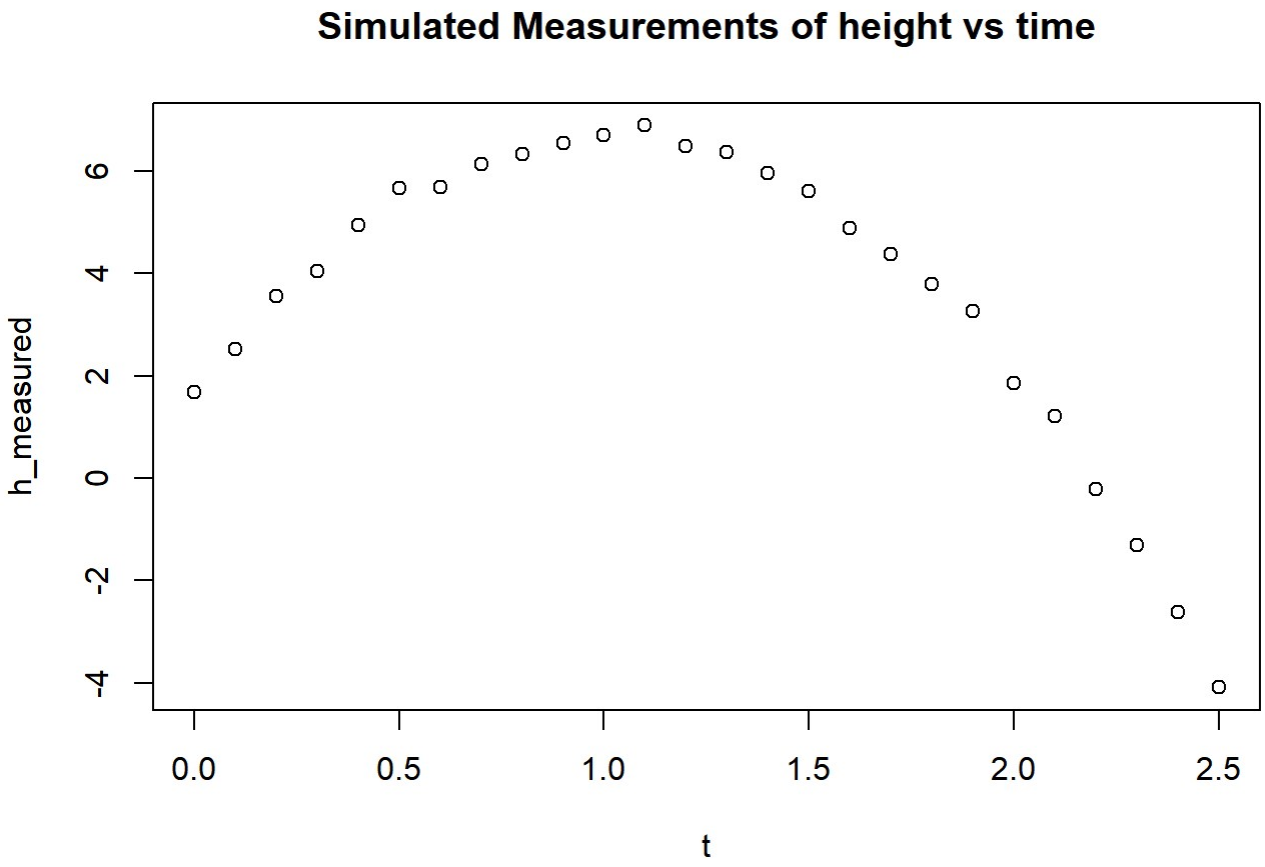
If we wanted to simulate the *measured* heights, instead of the predicted heights, we could take our list of predicted heights and add an *error* term e, where e is a random normal value with a mean of zero and a standard deviation of 0.15. This would give us a *monte carlo* simulated version of an actual measurement,

rather than the “clean” theoretical simulation shown above

h_measured==1/2at^2+v0t+ h0+e

I’ll use the length function to create the same number of error terms as I have times

```
e_term=rnorm(length(t),mean=0,sd=0.15)
h_measured=0.5*g*t^2+v0*t+h0+e_term
plot(t,h_measured,type="p",main="Simulated Measurements of height vs time")
```



###Selecting, Filtering and Subsetting a vector

In many cases, we would like to select some subset of a vector

Let’s work with the t and h_measured values from the throwm object example

We can specify a *slice* of the data by using a vector to specify which elements of an other vector we want to extract, using a square bracket

```
t[1]
```

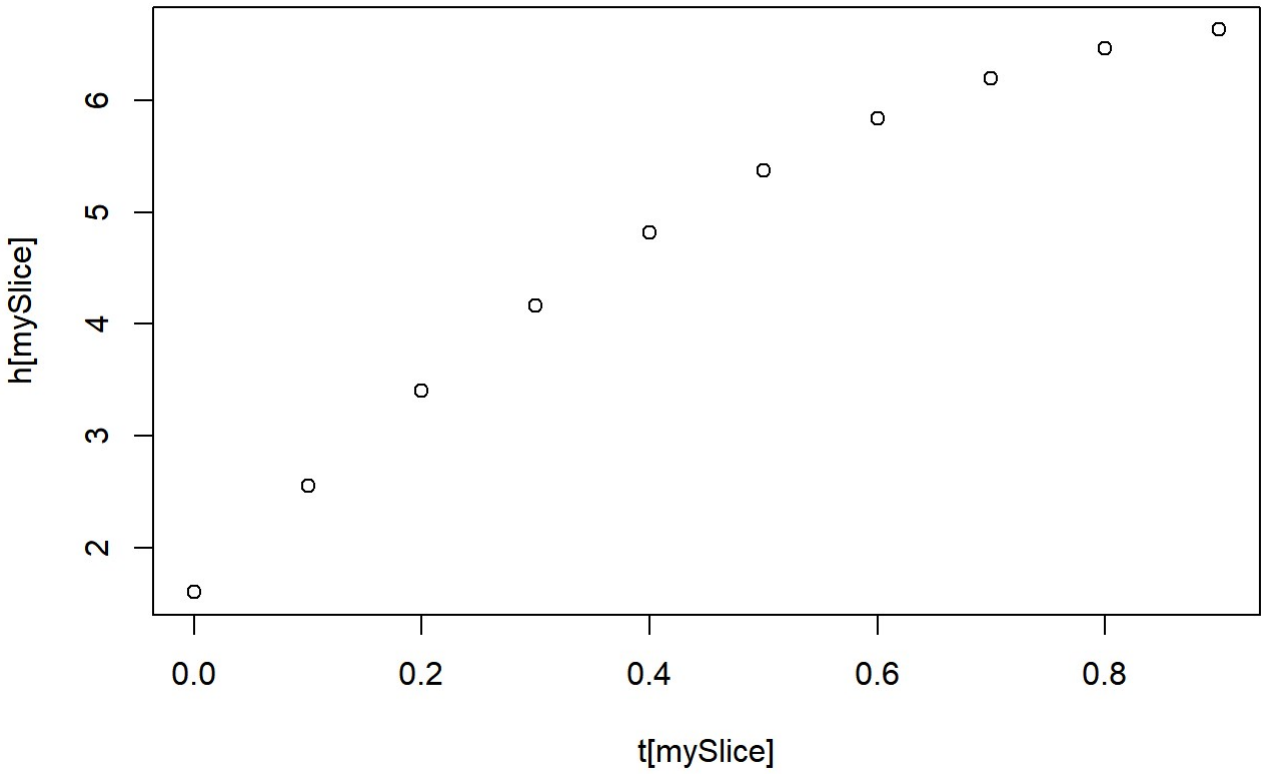
```
## [1] 0
```

We can do this with a range of values, lets extract the first 10 time values

```
t[1:10]
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9
```

```
mySlice=1:10
plot(t[mySlice],h[mySlice],type='p')
```



We could extract some other slices or subsets, say I wanted the 1st, 3rd and 7th heights for some odd reason, I could use a vector of the values 1,3,7 to do the slicing

```
t[c(1,3,7)]
```

```
## [1] 0.0 0.2 0.6
```

###Indexing using Binary Arrays

We can also use a binary array to select or slice. This seems like a pretty odd idea, but it is incredibly useful. Remember a binary array is just an arry of TRUE and FALSE values.

To index a vector using a binary vector, the binary vector must have the same number of elements as the vector we want to index.

The height and time vectors we created above have 26 elements, so I need a 26 element binary array to use it

as an index on height and time. I'll create this binary array using the c() and rep() functions

```
binaryIndex=c(rep(TRUE,5),rep(FALSE,21))
binaryIndex
```

```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
## [25] FALSE FALSE
```

We can now index t, and h_measured using this

```
t[binaryIndex]
```

```
## [1] 0.0 0.1 0.2 0.3 0.4
```

```
h_measured[binaryIndex]
```

```
## [1] 1.678989 2.521521 3.554086 4.050623 4.950898
```

Kinda cool, but setting up the array seems a bit tedious, since I just selected the first 5 points.

What we can do is generate the binary index vector using a test condition

Lets get a binary arrive of all the positions in h_measured where h is above zero, ie before it hits the group

```
pos_height=(h_measured>=0)
pos_height
```

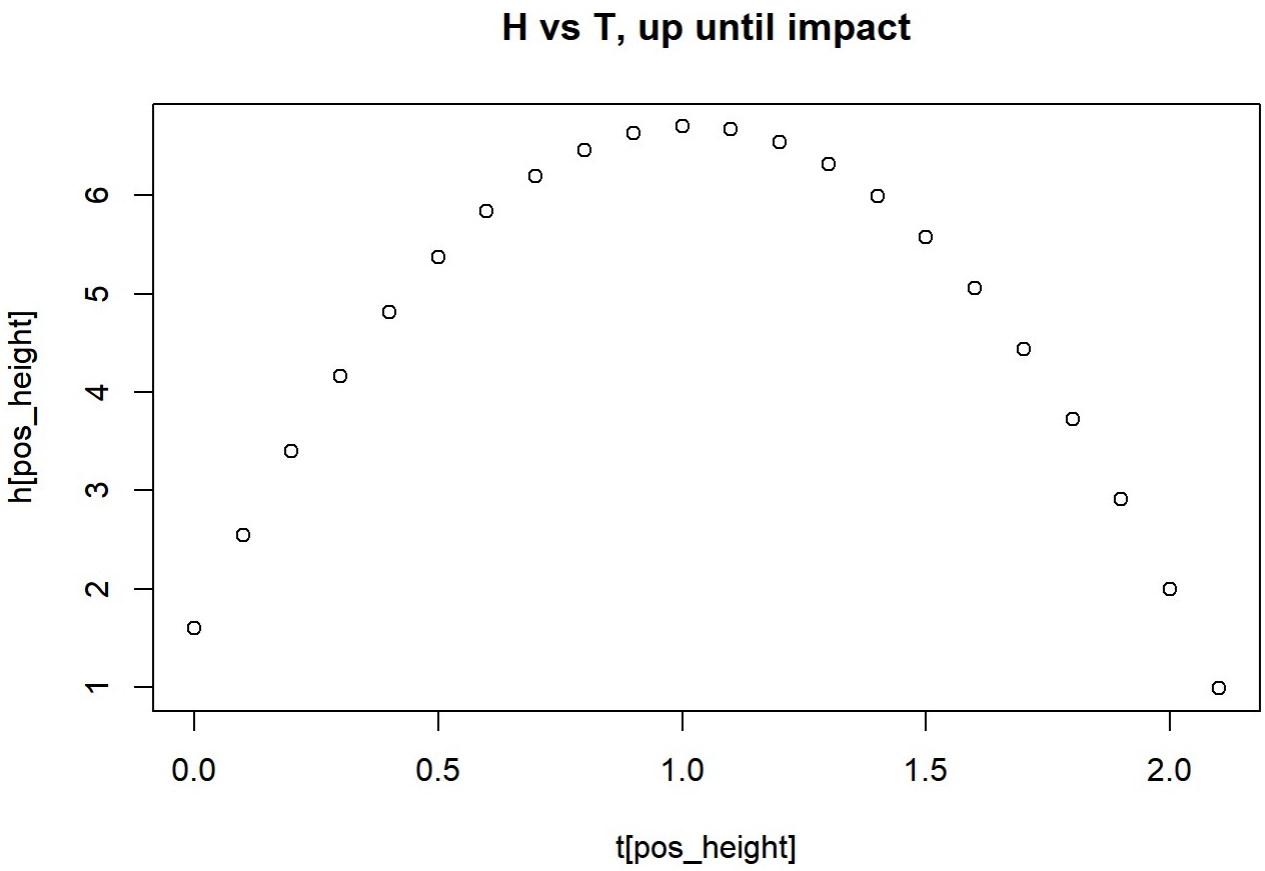
```
##  [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
## [13]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE FALSE FALSE
## [25] FALSE FALSE
```

We can use this to extract the subset of t and h_measured for which h_measured was above ground

```
t[pos_height]
```

```
##  [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8
## [20] 1.9 2.0 2.1
```

```
plot(t[pos_height],h[pos_height],type='p',main="H vs T, up until impact")
```



We could use the max() and min() functions to find the max and min values of height.

We could figure out which point is the maximum in the list of h values, and then index t to find the time of the maximum

```
mVec=(h_measured==max(h_measured))
t[mVec]
```

```
## [1] 1.1
```

###Combining logical conditions

There are the standard logical functions AND, OR and NOT available written as

a|b- a OR b a&b- a AND b ~a - NOT a

I'm using the cat() function to concatenate the output here, to make them easier to read

```
a=TRUE
b=TRUE
c=FALSE
cat("a or b:", a|b,"\n")
```

```
## a or b: TRUE
```

```
cat("a and b:",a&b,"\n")
```

```
## a and b: TRUE
```

```
cat("a and c", a&c,"\n")
```

```
## a and c FALSE
```

```
cat("Not (a and c)",!(a&c),"\n")
```

```
## Not (a and c) TRUE
```

Let's look for time values after the maximum height at t=1.1 second, but where the height is greater than zero

```
mySlice=(t>1.1)&(h_measured>0)
t[mySlice]
```

```
##  [1] 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1
```

The which() function

We can also generate a list of the indices of positions which meet at given condition, instead of using an entire binary array- it can be easier to read the list

```
my_animals=c('cat','dog','pony','cat','hamster')
names(my_animals)=c("Fluffy","Buster","Trigger","Bob","Pansy")
cat_index=which(my_animals=='cat')
names(my_animals[cat_index])
```

```
## [1] "Fluffy" "Bob"
```

```
cat("\n")
```

```
cat_index
```

```
## Fluffy    Bob
##      1      4
```

We can also index an array by name

```
my_animals["Trigger"]
```

```
## Trigger
##  "pony"
```

This typing of “slicing” or “subsetting” is key to cleaning data, and extracting information from files.

Think about how you can do this in Excel or in a SQL data base (using a Query), and how this compares to the appraoch in R

Note: Python has many of the same types of query functions