Summary:

Simple expressions, vectors

> 10:-1
> (9:-9)^2
>

Variables

> a=1:9
> a^3
>

Functions

> a
> length(a)
> sum(a)
> min(a)
> max(a)
> 

How to get help

There are two major ways to get help in R. The first, is calling a browser with all the documentation

> help.start()
>

The second is from within R:

> ?length
>

There are many other ways to find help, such as

> apropos("length")
> ?apropos
> args(length)
> args(min)
>

More on vectors

> a=2:11
> a
>

How can we access a certain entry in a?
In square brackets we write the entry we want to see.

We can also set part of the vector:

```r
> a
> a[3]
> a[3]=100
> a
```

But, as we said, R doesn’t really work with single number, but with vectors. This is also true here. Instead of asking for a single entry, we can ask for a couple

```r
> a[1:3]
> a[1:5]

and also

> a[3:1]
> a[10:1]
```

And, we can set a part of the vector:

```r
> a[1:3] = c(9,5,4)
> a
```

We can also ask to exclude some of the entries in a vector

```r
> a=10:1
> a
> a[-1]
> a[-(1:5)]
> a[1:5]
> a[c(3,3,3,4)]
```

We can’t mix positive and negative entries

```r
[a[<-1,2]]
```

```
```
Other types of vectors

There are several types of vectors in R. Two we already encountered

1. numbers (reals)
2. strings (character)
3. numbers (integers)
4. boolean
5. factors
6. complex

[0m> 1:10
> c("a","b","c")
>

Let us look at booleans

> c(TRUE,TRUE,FALSE)
> c(T,T,F)
> a
> a>5
> i= a>5
> i
> a[i]
>

Thus, we can subset by a boolean vector.

> a=1:10
> a[a>3 & a<8]
> a[ a>3] = 7
> a
> a[a>3] = 1:2

[0m> a
> a[a==1] = (1:10)*100
[0m> a
[0m>

Random sampling

Let us say we want to throw a die. 6 sides, all have equal chance

[0m> sample(1:6,1)
> sample(1:6,1)
> sample(1:6,1)
> sample(1:6,1)
> sample(6,1)
> sample(6,1)
> sample(1:6,2)
Now we have a function that accepts two parameters: what we sample from, and how many samples we want.

Now, we’d like to use what we learned. I would like to take a population of individuals, and ask which of them will have offspring.

```r
population=1:10
population
```

Let us say 3 have offspring, which 3?

```r
sample(population,3)
sample(population,3)
```

What if every offspring comes from one parent. And a parent is randomly chosen for each offspring?

```r
offspring=sample(population,10)
offspring
```

What did we get?

```r
sort(offspring)
```

The way we sampled now, each parent is chosen for one and only one offspring. We want each parent to be a candidate for each offspring, no matter if it was already a parent for someone else.

Let us look at the explanation of `sample()`

```r
?sample
sample(population,10,replace=T)
```

This is a nice feature of R. We have what is called “named parameters”, and default parameters. With named parameters, we don’t need to remember the order, and because there are default parameters, we don’t need to specify all of them.

```r
offspring=sample(population,10,replace=T)
offspring
sort(offspring)
```

This is called the “Wright-Fisher model of reproduction”. Every individual can have from 0 to N offspring.

It is, of course not the only way we can model reproduction. We can imagine something else: Say, every individual has 3 offspring, and of those, only N survive

```r
offspring = c(population,population,population)
```
In this case, each parent has 0 to 3 offspring. There are, of course, differences between these!

> p=1:10
> p=sample(p,10,rep=T)
> sort(p)

Notice that we lost some individuals. Those we will never get back! Let us do another generation

> p=sample(p,10,rep=T)
> sort(p)

We lost some more...

> p=sample(p,10,rep=T); sort(p)
> p=sample(p,10,rep=T); sort(p)
> p=sample(p,10,rep=T); sort(p)
> p=sample(p,10,rep=T); sort(p)
> p=sample(p,10,rep=T); sort(p)
> p=sample(p,10,rep=T); sort(p)

Now, let us try the same, where every individual has 0 to 2 offspring.

> p=1:10
> p=sample(c(p,p), 10); sort(p)

This took longer. The reason is that the number of offspring different individuals have is more similar, so it takes longer for frequencies to change.

> p=1:100
> o=sample(p,100,rep=T)
> sort(o)

I’d like to count how many offspring each parent had

> sort(table(o))

Let us count these

> table(table(o))
> sum( table(table(o)))
>
> plot(table(table(o))); v()
Let us try the same with the 2 offspring model:

```r
> p=1:100
> o=sample( c(p,p), 100)
> o
> table(o)
> table(table(o))
> plot( table(table(o)) ) ; v()
```

Let us do one more generation of the 2 offspring type

```r
> p=o
> o=sample( c(p,p), 100)
> plot( table(table(o)) ) ; v()
```

In some sense, one can say that you need more generations of the 2nd type to get the same effect as the first type.

How much more? Let us try an even bigger population

```r
> p=1:10^6
> o1=sample(p,length(p), rep=T)
> o2=sample( c(p,p), length(p), rep=F)
> hist(table(o1), breaks=(0:10)-0.5); v()
> hist(table(o2), breaks=(0:10)-0.5); v()
> var(table(o1))
> var(table(o2))
> sum(o1 <= 5e5)-5e5
> sum(o2 <= 5e5)-5e5
```

Let us say we wanted to track just the last two individuals. There is another way to accomplish what we want:

```r
> p=1:10^6
> p[p <= 500000]=1
> p[p > 500000] = 2
> table(p)
> p1=p
> o1=sample(p1, length(p1), rep=T)
> table(o1)
```

But we don’t really need to sample from 500,000 1 and 500,000 2 with replacement. This is the same as sampling from 1:2 with replacement

```r
> o1=sample(1:2,10^6,rep=T)
```
But what happens when the numbers aren’t exactly 50:50? The we can ask to sample with certain probabilities:

```r
> sample(1:2, 20, rep=T, prob=c(0.1, 0.9))
[1] 2 2 2 2 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

This is the 4th parameter of `sample()` that we use.

We can specify the probabilities. So, now we can do the whole thing again:

```r
> table(p)
> o1=sample(1:2, 10^6, rep=T, p=c(0.5, 0.5))
> table(o1)
> o2=sample(1:2, 10^6, rep=T, p=table(o1)/10^6)
> table(o2)
> o3=sample(1:2, 10^6, rep=T, p=table(o2)/10^6)
> table(o3)
> table(p)
```