CT5102: Programming for Data Analytics

Week 2:
Functions in R
Lists

https://github.com/JimDuggan/CT5102

Dr. Jim Duggan,
Information Technology,
School of Engineering & Informatics

Functions

- A function is a group of instructions that:
  - takes input,
  - uses the input to compute other value, and
  - returns a result (Matloff 2009).
- Users of R should adopt the habit of creating simple functions which will make their work more effective and also more trustworthy (Chambers 2008).
- Functions are declared:
  - using the **function** reserved word
  - are objects

General Form

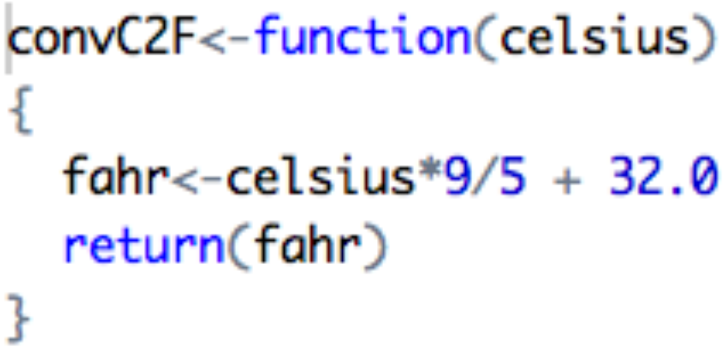
function(*arguments*)

    expression

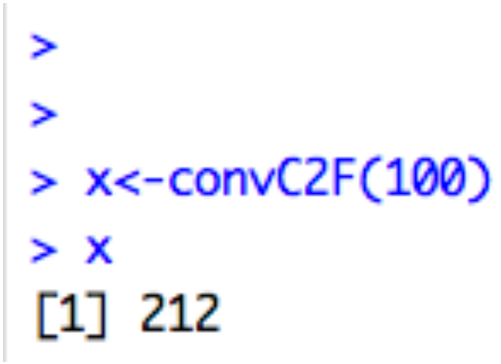
- *arguments* gives the arguments, separated by commas.
- *Expression* (body of the function) is any legal R expression, usually enclosed in { }

Example

- Function declared as an object
- Takes arguments
- Local scope for function variables
- Generates result
- return() explicit
- Implicit – last expression evaluated.



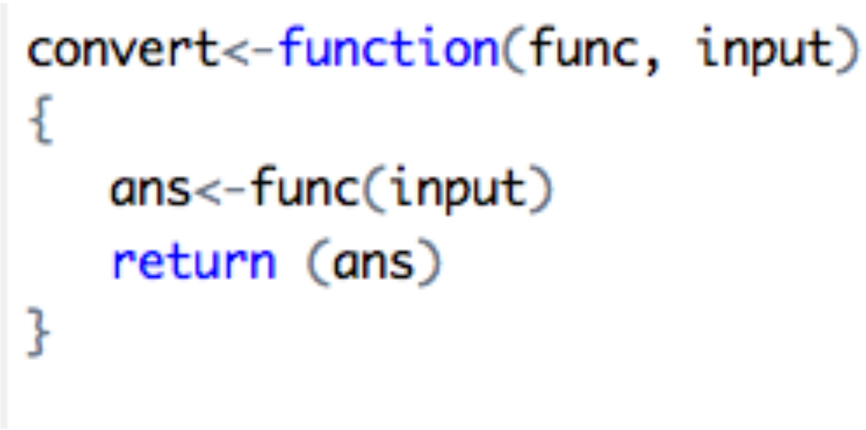
```
convC2F<-function(celsius)
{
  fahr<-celsius*9/5 + 32.0
  return(fahr)
}
```



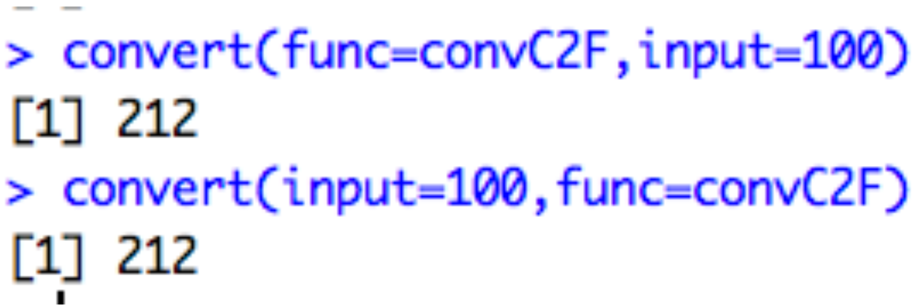
```
>
>
> x<-convC2F(100)
> x
[1] 212
```

Function objects and arguments

- Functions are first-class objects, can be passed to other functions
- Arguments can be named directly in the call
- In that scenario, order of arguments not an issue



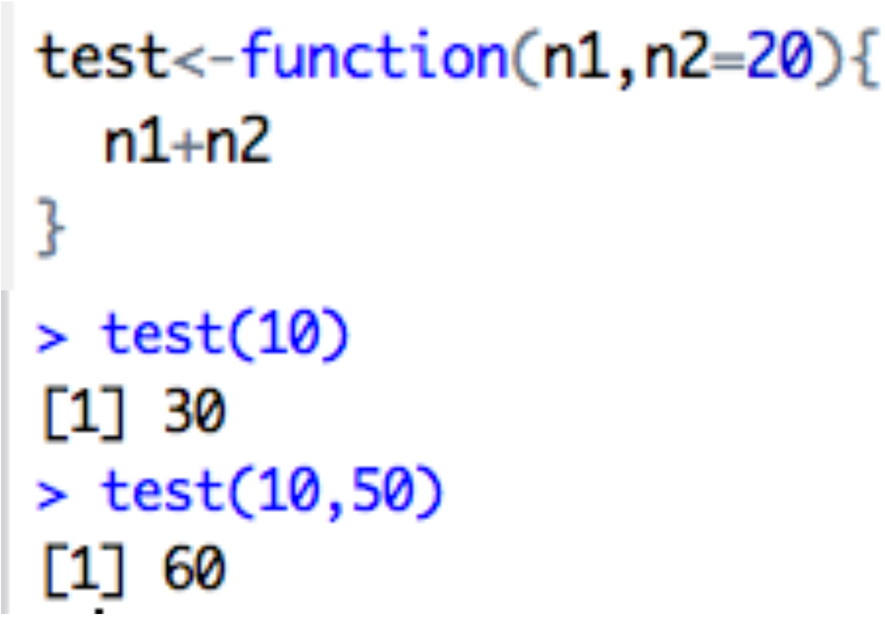
```
convert<-function(func, input)
{
   ans<-func(input)
   return (ans)
}
```



```
> convert(func=convC2F,input=100)
[1] 212
> convert(input=100,func=convC2F)
[1] 212
```

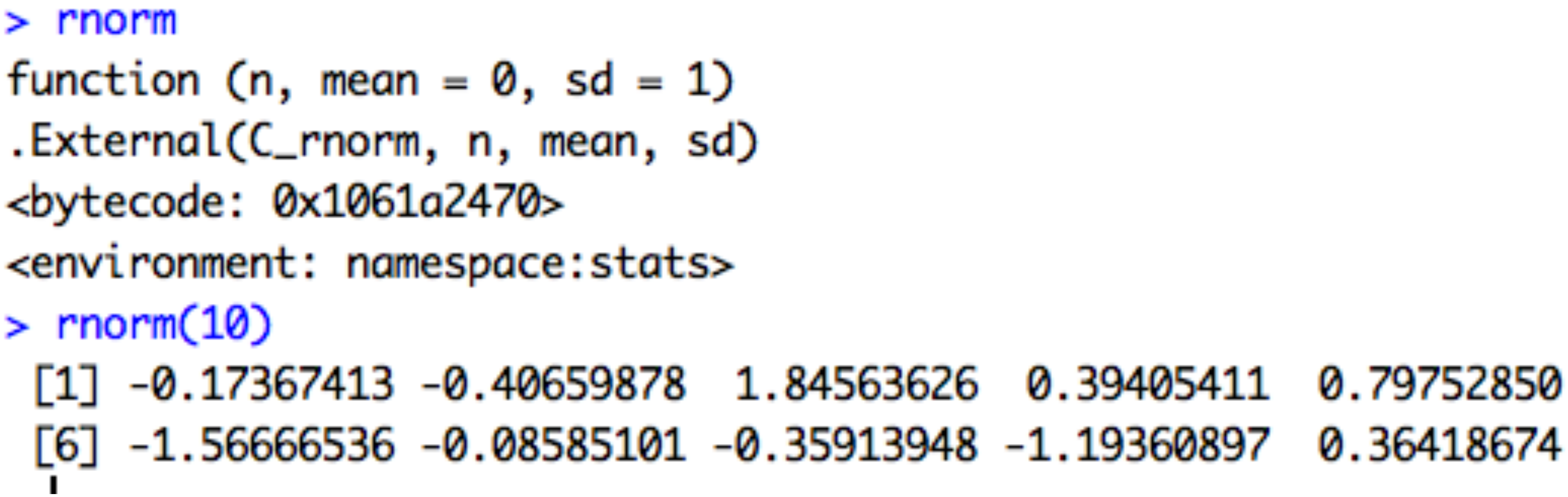
Default values

- Assigned when the argument declared
- Useful to have default values



```
test<-function(n1,n2=20){
  n1+n2
}
```

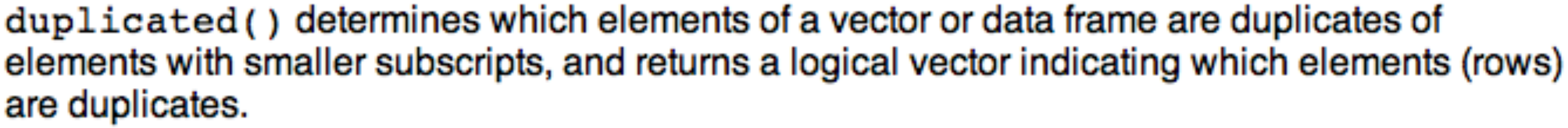
```
> test(10)
[1] 30
> test(10,50)
[1] 60
```



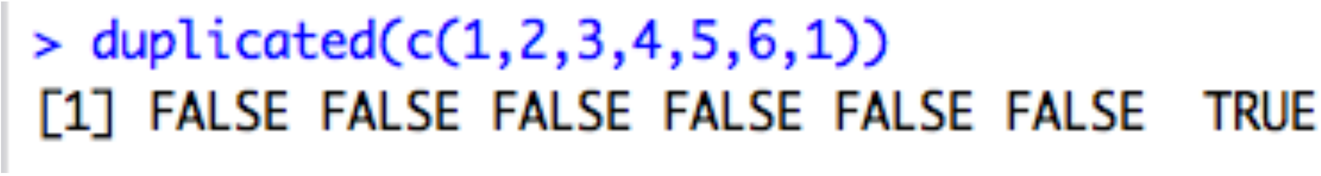
```
> rnorm
function (n, mean = 0, sd = 1)
.External(C_rnorm, n, mean, sd)
<bytecode: 0x1061a2470>
<environment: namespace:stats>
> rnorm(10)
 [1] -0.17367413 -0.40659878  1.84563626  0.39405411  0.79752850
 [6] -1.56666536 -0.08585101 -0.35913948 -1.19360897  0.36418674
```

Challenge 2.1
One line functions

- Write a function that takes in a vector and returns the number of odd numbers
- Write a function that takes in a vector and returns a vector of even numbers
- Write a function that returns the unique values in a vector



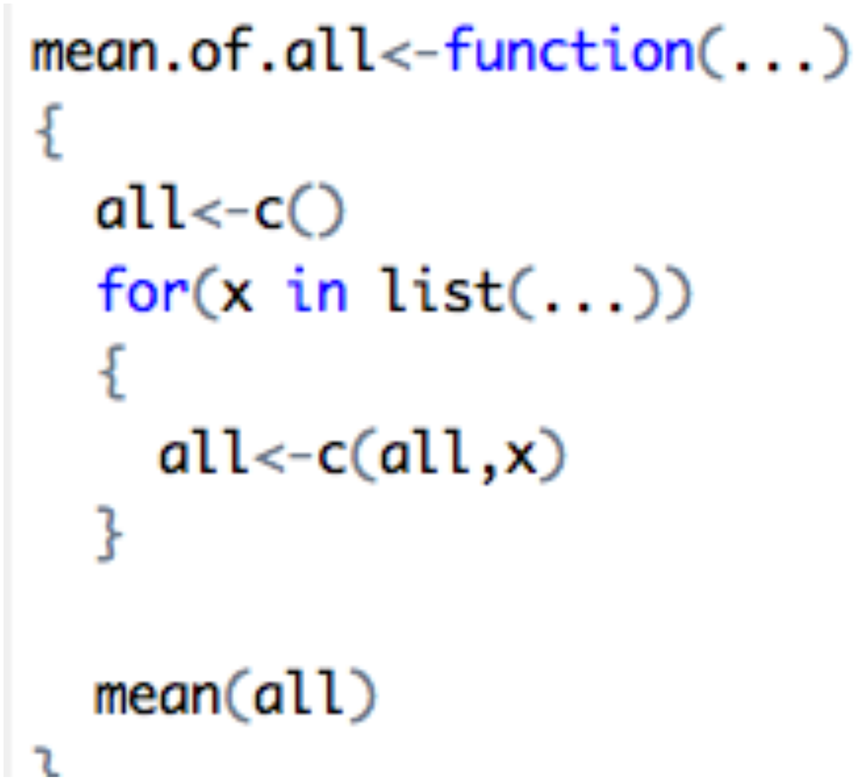
`duplicated()` determines which elements of a vector or data frame are duplicates of elements with smaller subscripts, and returns a logical vector indicating which elements (rows) are duplicates.



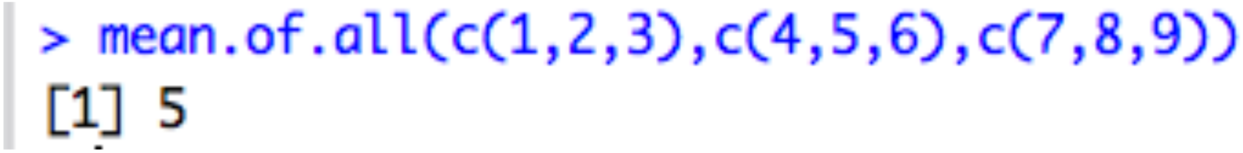
```
> duplicated(c(1,2,3,4,5,6,1))
[1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
```

Variable number of arguments

- Functions can take a variable number of arguments
- The special argument name is "…"
- This can be converted to a list and processed in the function (more on lists later)



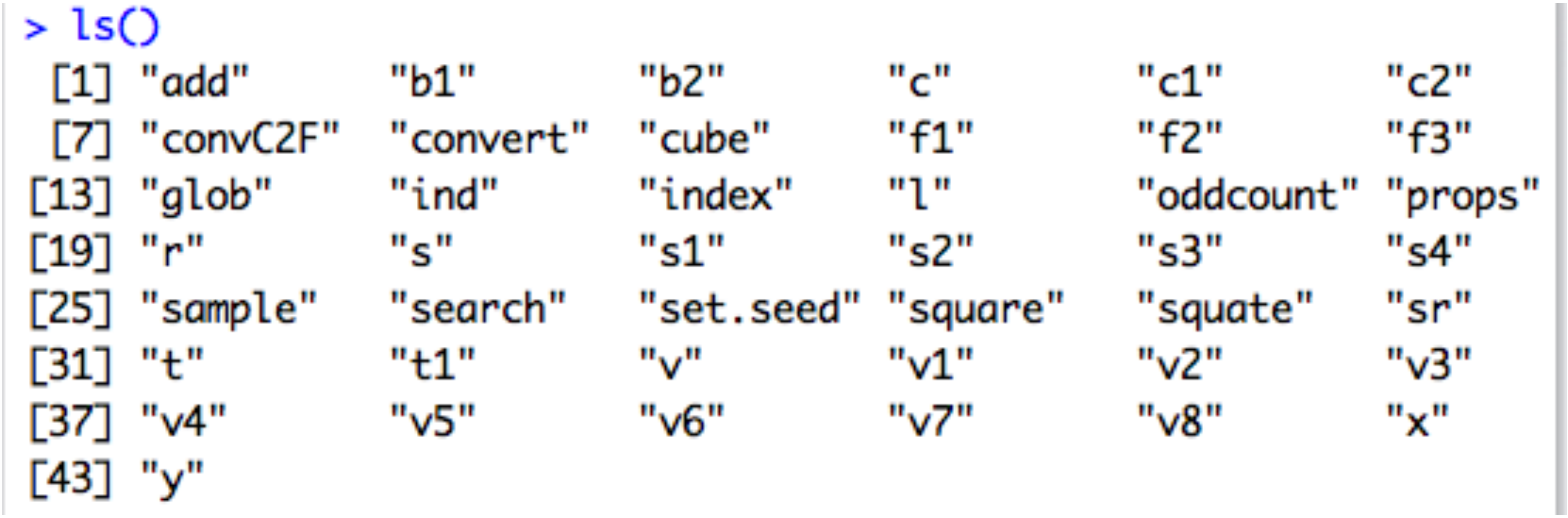
```
mean.of.all<-function(...)
{
  all<-c()
  for(x in list(...))
  {
    all<-c(all,x)
  }

  mean(all)
}
```



```
> mean.of.all(c(1,2,3),c(4,5,6),c(7,8,9))
[1] 5
```

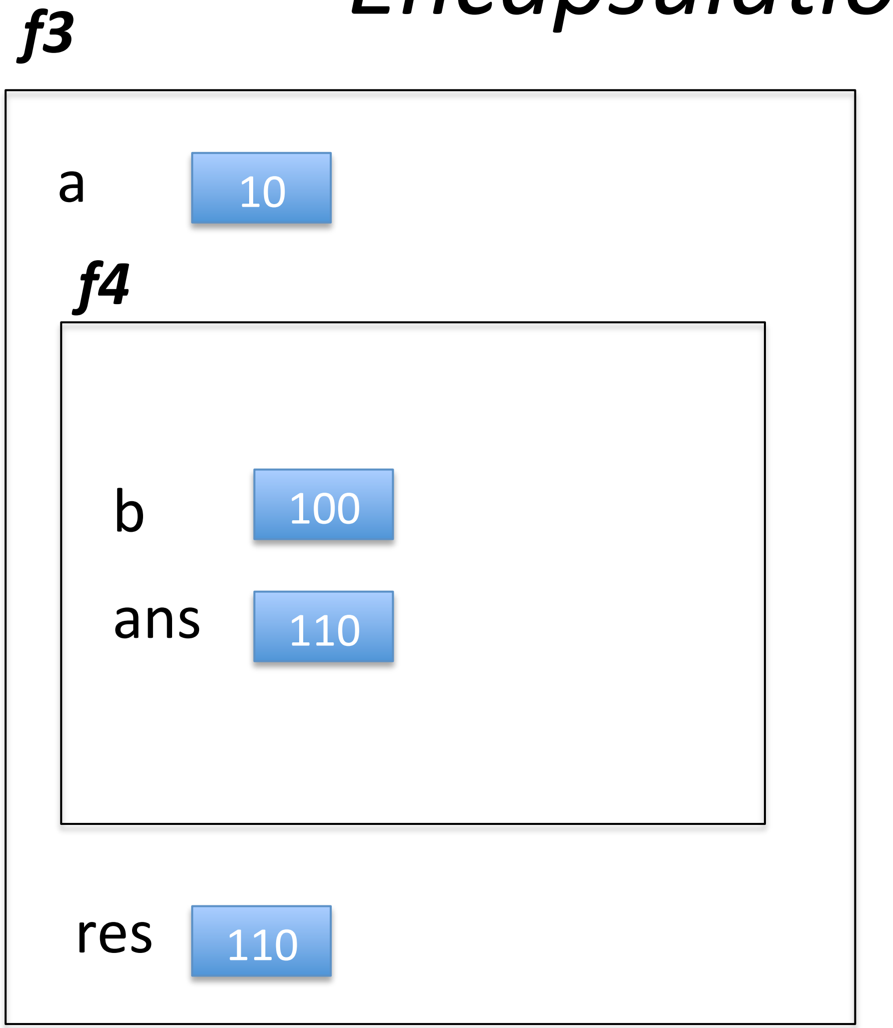
Environment and Scope Issues

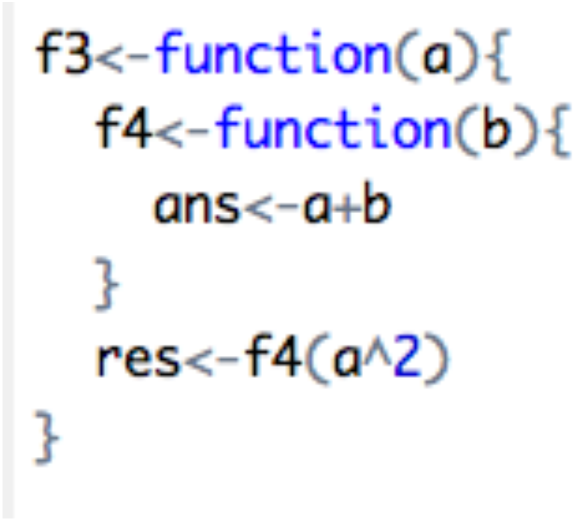
- A function consists of its arguments and body AND its environment
- These are the collection of objects present when the function is created
- Top level environment is the GlobalEnvironment



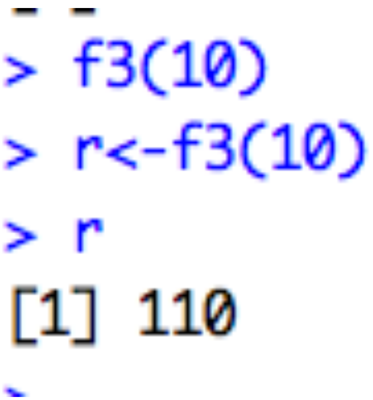
```
> ls()
 [1] "add"      "b1"       "b2"       "c"        "c1"       "c2"
 [7] "convC2F"  "convert"  "cube"     "f1"       "f2"       "f3"
[13] "glob"     "ind"      "index"    "l"        "oddcount" "props"
[19] "r"        "s"        "s1"       "s2"       "s3"       "s4"
[25] "sample"   "search"   "set.seed" "square"   "squate"   "sr"
[31] "t"        "t1"       "v"        "v1"       "v2"       "v3"
[37] "v4"       "v5"       "v6"       "v7"       "v8"       "x"
[43] "y"
```

Nested Functions:
Encapsulation Hierarchy





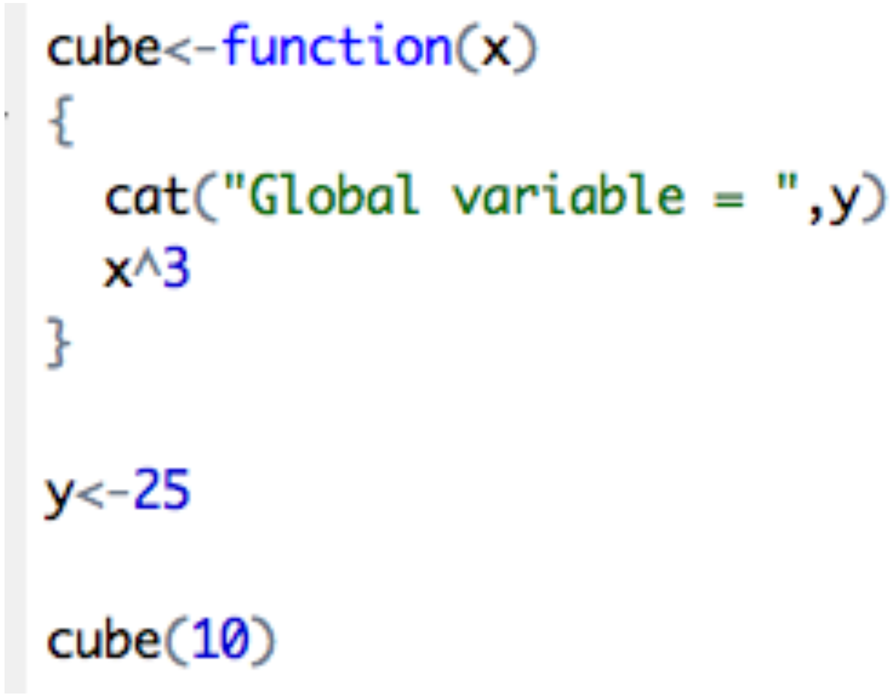
```
f3<-function(a){
   f4<-function(b){
      ans<-a+b
   }
   res<-f4(a^2)
}
```



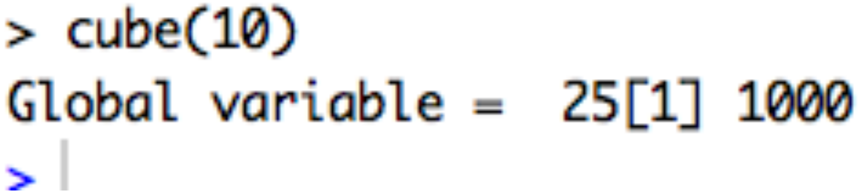
```
> f3(10)
> r<-f3(10)
> r
[1] 110
```

Variable Scope

- A variable that is visible within a function body is local to that function
- Formal parameters are local variables
- Variables created outside a function are global and available to that function



```
cube<-function(x)
{
  cat("Global variable = ",y)
  x^3
}

y<-25

cube(10)
```



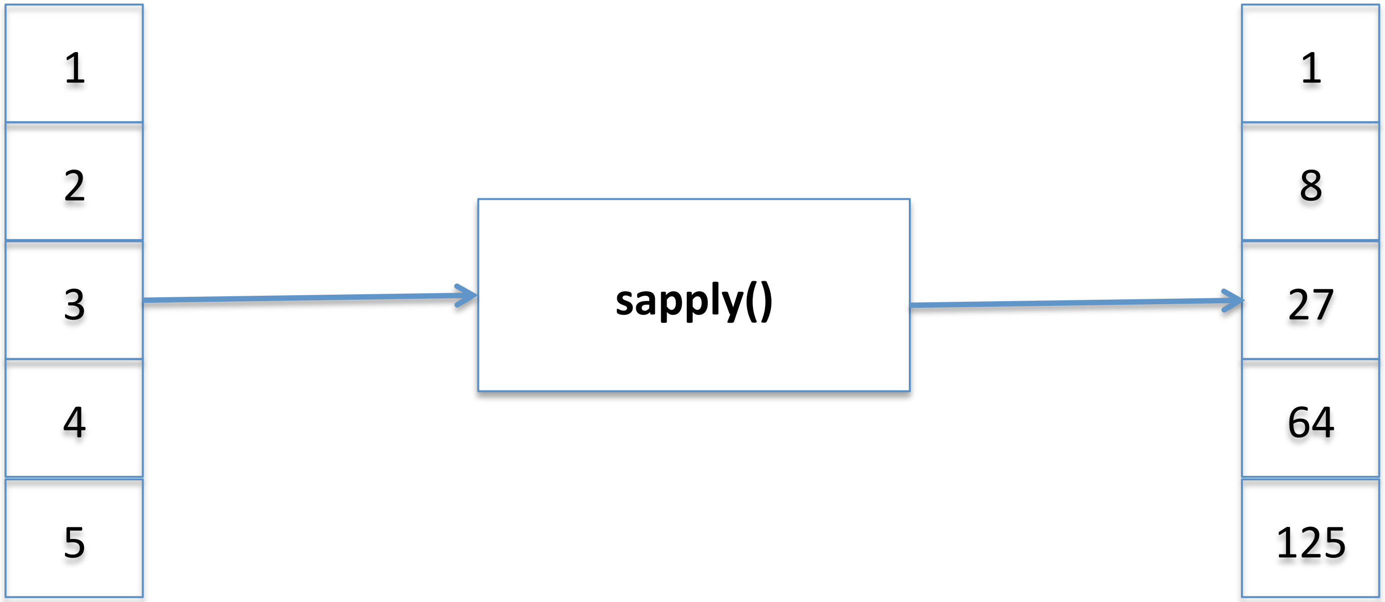
```
> cube(10)
Global variable =  25[1] 1000
> 
```

sapply()

- Another use of user-defined functions in R is as parameters to the *apply* family of functions, which are one of the most famous and used features of R (Matloff 2009).
- The general form of the **sapply(x,f,fargs)** function is as follows:
  - **x** is the target vector or list
  - **f** is the function to be called and applied to each element
  - **fargs** are the optional set of arguments that can be applied to the function **f**.

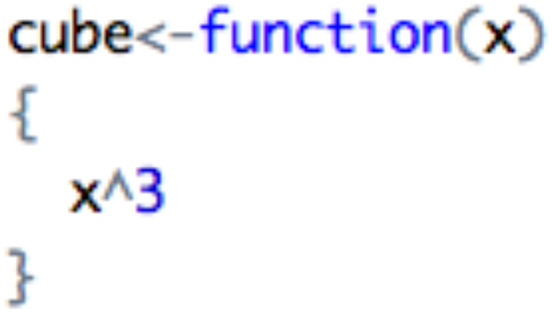
Sample Problem

- To get the cube of numbers in a vector

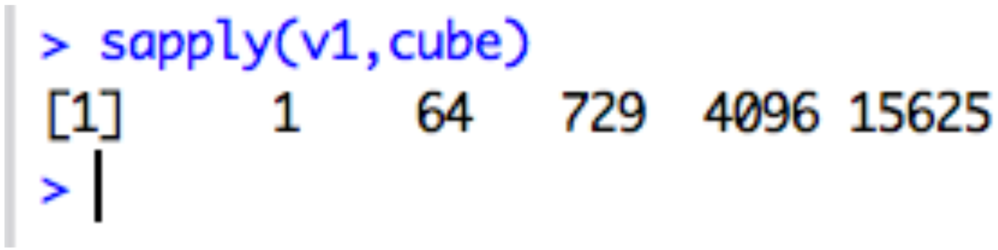


Method 1

- Write separate function
- Input vector as first argument
- Function as second argument



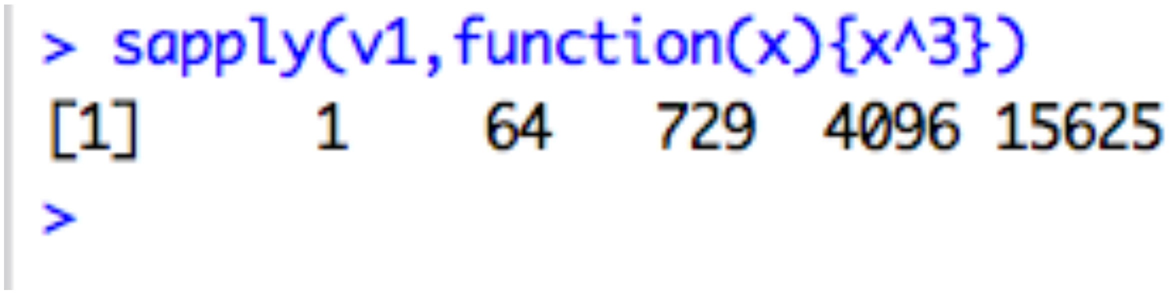
```
cube<-function(x)
{
  x^3
}
```



```
> sapply(v1,cube)
[1]     1    64   729  4096 15625
> 
```

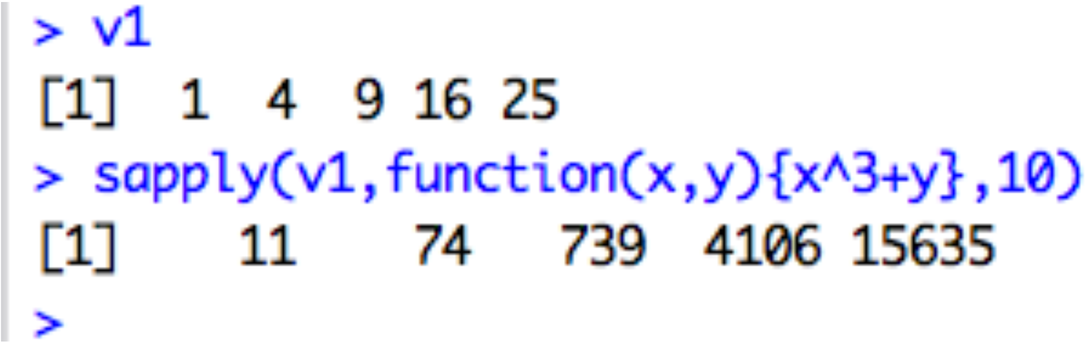
Method 2

- Embed function in call



```
> sapply(v1,function(x){x^3})
[1]     1    64   729  4096 15625
>
```

Add parameters



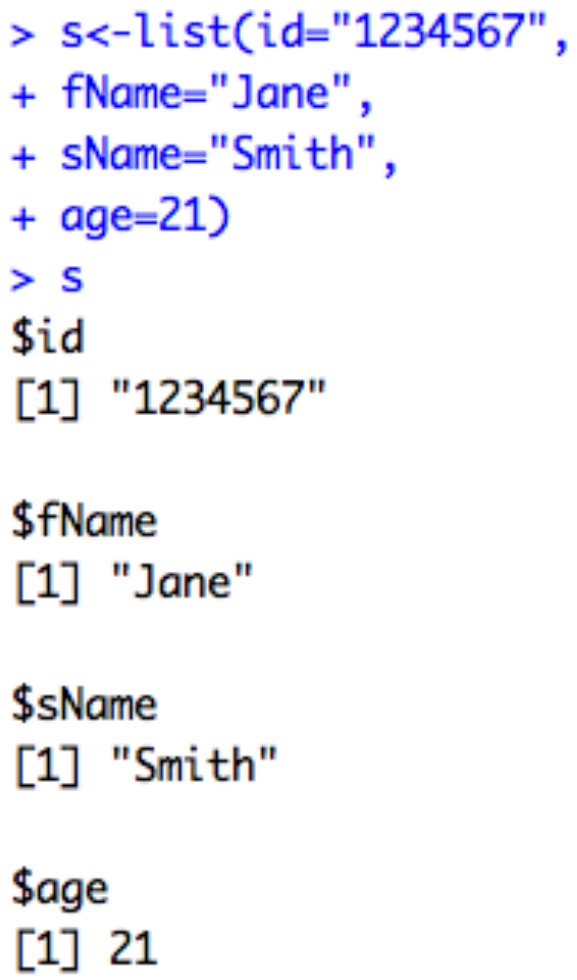
```
> v1
[1]  1  4  9 16 25
> sapply(v1,function(x,y){x^3+y},10)
[1]    11    74   739  4106 15635
>
```

Challenge 2.2

- Use sapply() for the following tasks
  - to process a vector and return the absolute difference of every element from the mean
  - transform a vector according to the following function
    - f(x) = $3x^2+5x+10$
- Plot the answer to the 2nd part using the plot() function

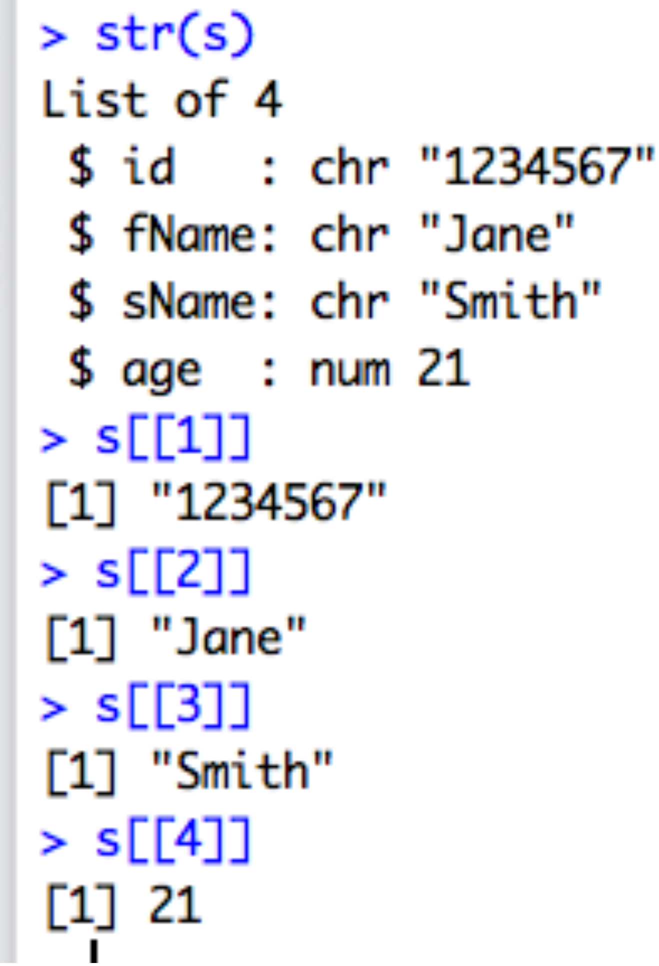
List Data Structure

- Unlike a vector, where all elements must be of the same mode, R's list structure can combine objects of different types (Matloff 2009).
- For example, using the **list()** function, we could create a list to represent information on a student.

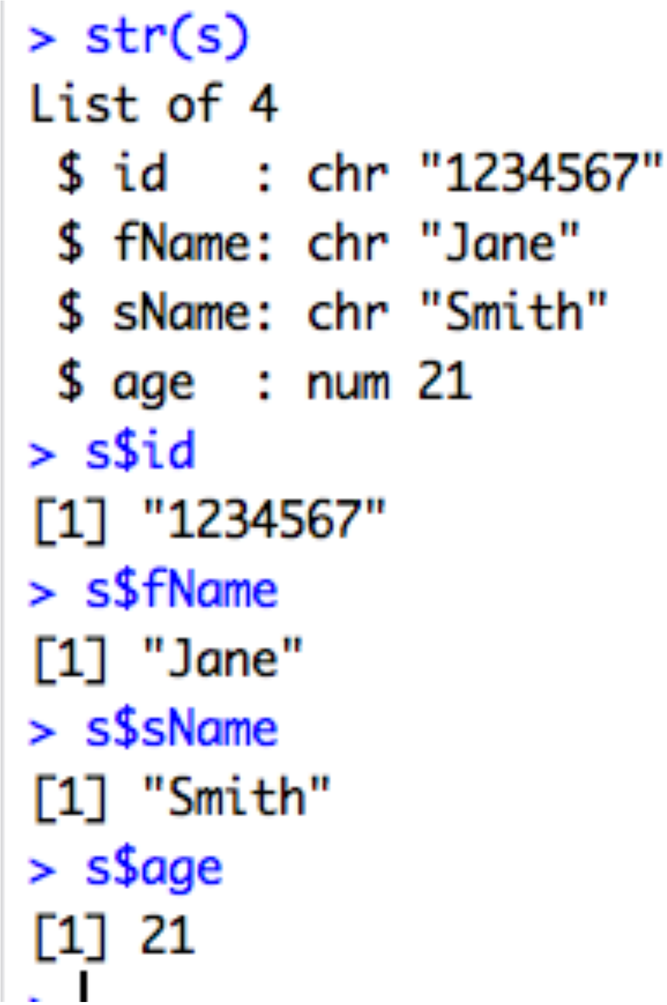


```
> s<-list(id="1234567",
+ fName="Jane",
+ sName="Smith",
+ age=21)
> s
$id
[1] "1234567"

$fName
[1] "Jane"

$sName
[1] "Smith"

$age
[1] 21
```

Accessing list elements

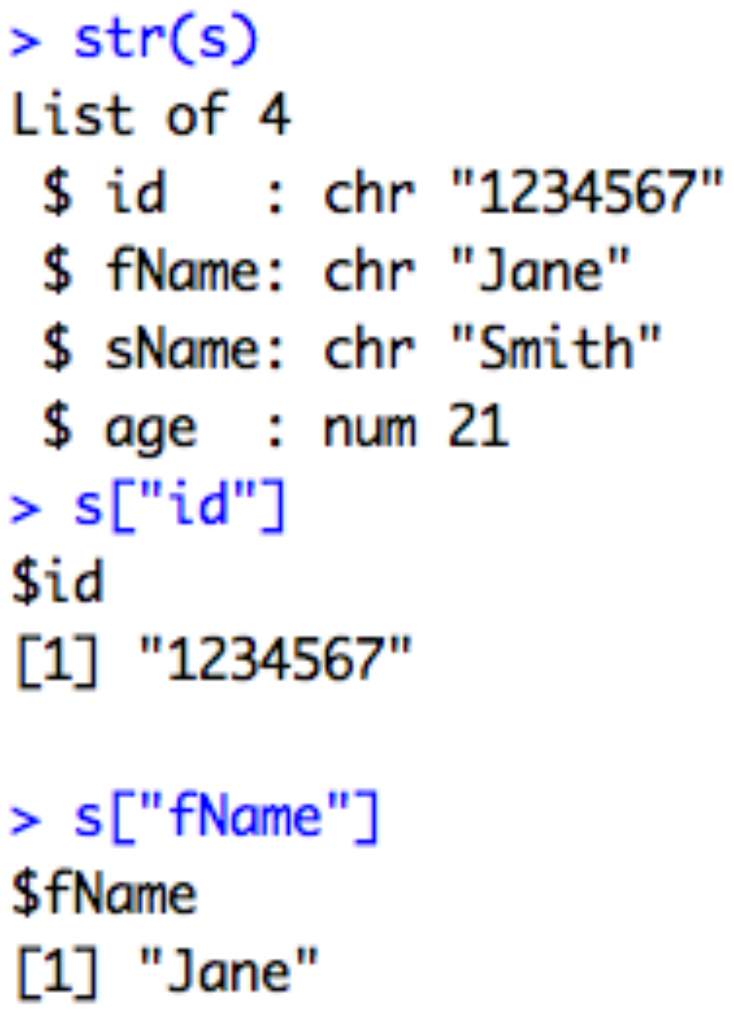


```
> str(s)
List of 4
 $ id   : chr "1234567"
 $ fName: chr "Jane"
 $ sName: chr "Smith"
 $ age  : num 21
> s[[1]]
[1] "1234567"
> s[[2]]
[1] "Jane"
> s[[3]]
[1] "Smith"
> s[[4]]
[1] 21
```

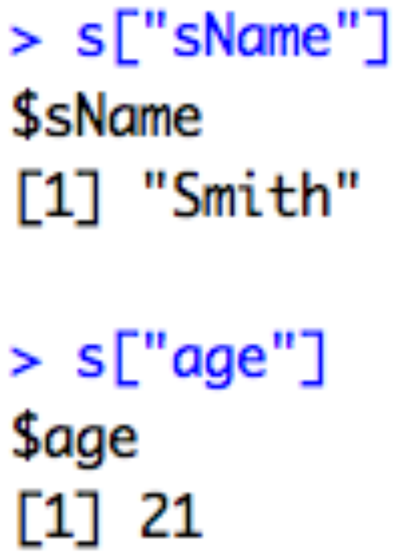


```
> str(s)
List of 4
 $ id   : chr "1234567"
 $ fName: chr "Jane"
 $ sName: chr "Smith"
 $ age  : num 21
> s$id
[1] "1234567"
> s$fName
[1] "Jane"
> s$sName
[1] "Smith"
> s$age
[1] 21
```

Accessing elements

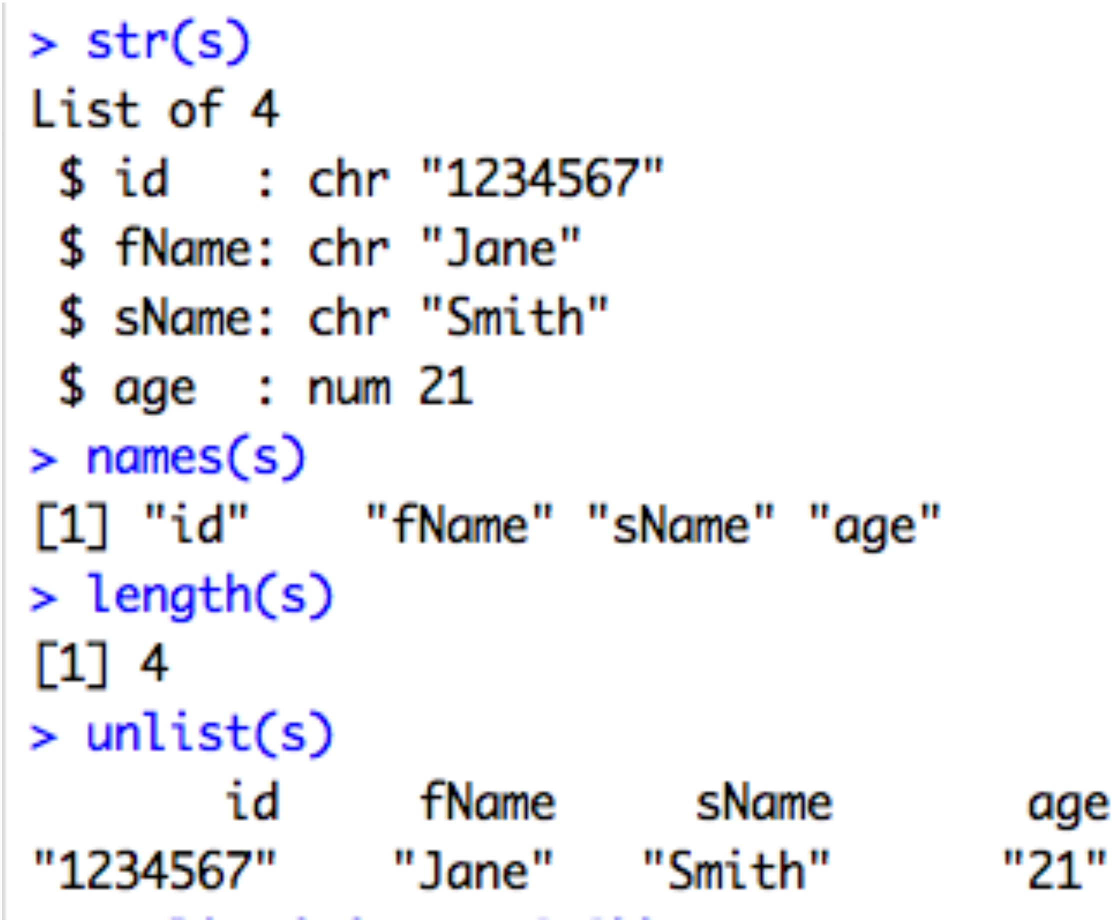


```
> str(s)
List of 4
 $ id   : chr "1234567"
 $ fName: chr "Jane"
 $ sName: chr "Smith"
 $ age  : num 21
> s["id"]
$id
[1] "1234567"

> s["fName"]
$fName
[1] "Jane"
```



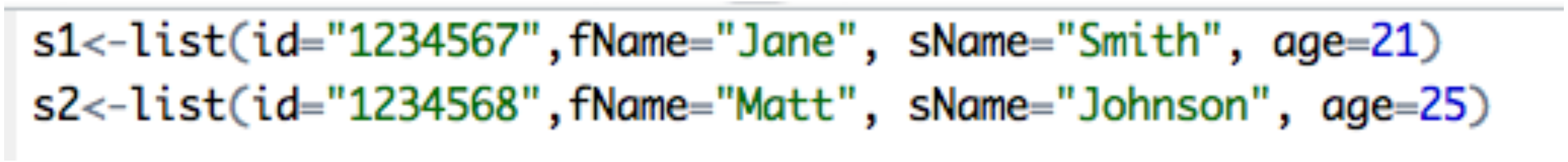
```
> s["sName"]
$sName
[1] "Smith"

> s["age"]
$age
[1] 21
```

More list operations

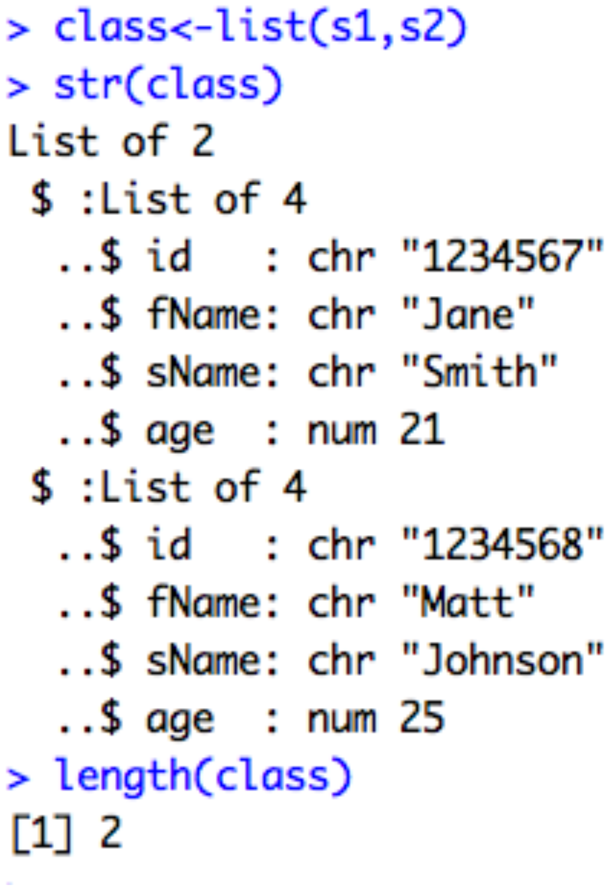


```
> str(s)
List of 4
 $ id   : chr "1234567"
 $ fName: chr "Jane"
 $ sName: chr "Smith"
 $ age  : num 21
> names(s)
[1] "id"    "fName" "sName" "age"
> length(s)
[1] 4
> unlist(s)
       id     fName      sName        age
"1234567"    "Jane"   "Smith"       "21"
```

Lists can contain lists…



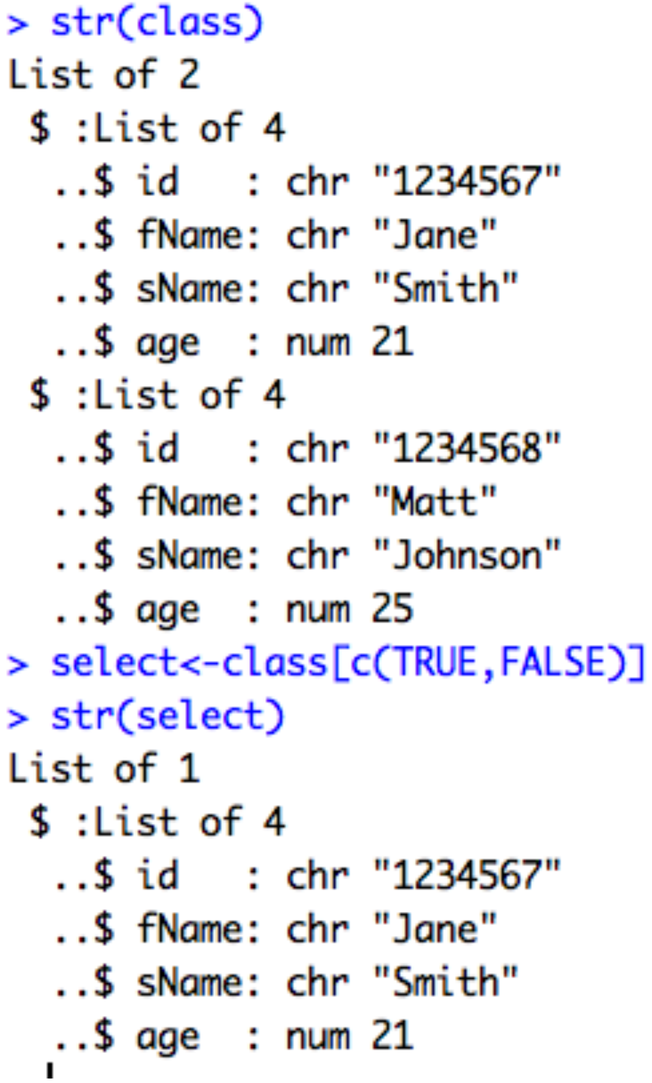
```
s1<-list(id="1234567",fName="Jane", sName="Smith", age=21)
s2<-list(id="1234568",fName="Matt", sName="Johnson", age=25)
```



```
> class<-list(s1,s2)
> str(class)
List of 2
 $ :List of 4
  ..$ id   : chr "1234567"
  ..$ fName: chr "Jane"
  ..$ sName: chr "Smith"
  ..$ age  : num 21
 $ :List of 4
  ..$ id   : chr "1234568"
  ..$ fName: chr "Matt"
  ..$ sName: chr "Johnson"
  ..$ age  : num 25
> length(class)
[1] 2
```

Filtering Lists

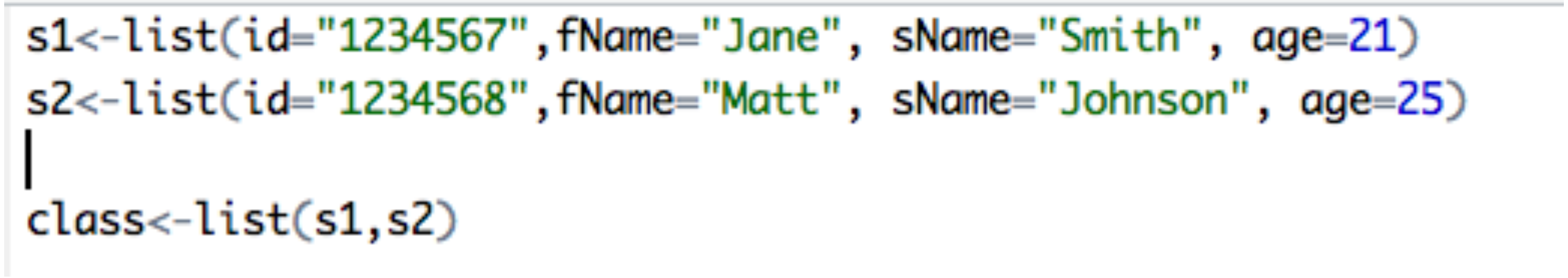
- Boolean vectors can be applied to lists, similar to vectors
- This can be used to filter elements



```
> str(class)
List of 2
 $ :List of 4
  ..$ id   : chr "1234567"
  ..$ fName: chr "Jane"
  ..$ sName: chr "Smith"
  ..$ age  : num 21
 $ :List of 4
  ..$ id   : chr "1234568"
  ..$ fName: chr "Matt"
  ..$ sName: chr "Johnson"
  ..$ age  : num 25
> select<-class[c(TRUE,FALSE)]
> str(select)
List of 1
 $ :List of 4
  ..$ id   : chr "1234567"
  ..$ fName: chr "Jane"
  ..$ sName: chr "Smith"
  ..$ age  : num 21
```

Challenge 2.3

- Write R code to find all those students in the list **class** whose age is greater than 21.
- Use the apply family of functions to find the matching list elements.



```
s1<-list(id="1234567",fName="Jane", sName="Smith", age=21)
s2<-list(id="1234568",fName="Matt", sName="Johnson", age=25)

class<-list(s1,s2)
```