
User functions

2

Introducing ... User functions

- All R commands are functions.
- Functions perform calculations, possibly involving several arguments, then return a value to the calling statement.
- The calculation maybe any process, might or might not have return value
 - It need not be arithmetic
- User functions extend the capabilities of R by adapting or creating new tasks that are tailored to your specific requirements.
- User functions are a special kind of object

Defining a new function

- Parts of function definition: name, arguments, procedural steps, return value

```
sqXplusX <- function(x){  
  x^2 + x  
}
```
- **sqXplusX** is the function name
- **x** is the single argument to this function and it exists only within the function
- everything between brackets { } are procedural steps
- the **last** calculated value is the function return value
- after defining the function, we can use it:

```
> sqXplusX(10)  
[1] 110
```

Named and default arguments

- Example of function with more than one named argument:

```
powXplusX <- function(x, power=2){  
  x^power + x  
}
```

- Now we have two arguments. The second argument has a default value of 2.
- Arguments without default value are required, those with default values are optional.

```
> powXplusX(10)  
[1] 110
```

```
> powXplusX(10, 3)  
[1] 1010
```

```
> powXplusX(x=10, power=3)  
[1] 1010
```

arguments matched based on **position**

arguments matched based on **name**

Assignments with arguments User functions

```
sqXplusX <- function(x){  
  x^2 + x  
}
```

You can use a blank document in gedit, nedit or other text editor to hold these commands for you, then copy / paste the instructions into R

- Now try this ...

```
a <- matrix(1:100, ncol=10, byrow=T) # make some dummy data  
b <- sqXplusX(a) # transform a by sqXplusX, assign result to b  
b # to view the result
```

- sqXplusX user function is now an R object, check its arguments and list it in the current workspace

```
> args(sqXplusX)  
> ls()
```

```
> sqXplusX
```

Don't add brackets to see the definition of sqXplusX

Assigned or anonymous ... User functions

- Functions may be assigned a name, or anonymously created within an operation
 - Anonymous functions are really useful in `apply()` style procedures

```
apply(object, margin, function)
```

- E.g. I have a 10 x 10 matrix and want to square each item, and add the item to itself

```
a <- matrix(1:100, ncol=10, byrow=T)  
a # to view new object
```

```
apply(a, c(1,2), function(x) x^2+x)
```

x is transiently assigned each item of a, and this is passed as an argument to the anonymous function

1 means by rows, 2 means by columns [1st or 2nd margin]
c(1,2) means do both rows and columns

Functions occupy their own space

User functions

- Objects created in functions are not available to the general environment unless returned.
 - they are said to be out of scope
 - Scope relates to the accessibility of an object.
- A function can only return one object.
- Custom functions disappear when R sessions end, unless the function object is saved in an Rdata file or sourced from a script.
 - A really useful function could be added to your .Rprofile file, and would always be ready for you at launch
- You could also make a package
 - Beyond the scope of the beginners course!!!!

Script / function tips

User functions

- If your script repeats the same style command more than twice, you should consider writing a function
- Writing functions makes your code more easily understandable because they encapsulate a procedure into a well-defined boundary with consistent input/output
- Functions should not be longer than one-to-two screens of code, keep functions clean and simple
- Look at other functions to get ideas for how to write your own ...
 - Display function code by entering the function's name without brackets.

File commands for extending scripts & user functions

Generic file commands

```
dir(..., pattern="txt")
```

Retrieve working directory file listing filtered by pattern. Note pattern is a regular expression, not a shell wildcard

```
glob2rx("*", txt)
```

Changes wildcards to regular expressions!

```
unlink(...)
```

Remove (permanently) a file from system

```
system(...)
```

Execute a shell command from within R

Result can not be coerced to an object, only available to linux R

```
> glob2rx("*", txt)
[1] "^.*\\.txt$"
```

Text manipulation for extending scripts & user functions

- Text manipulation and file name mangling ... that's a technical term

```
grep( pattern, object )
```

- If pattern is not found, grep returns a 0 length object.
 - Test for null with `is.null()`

```
sub( pattern, replacement, object )
```

```
gsub( pattern, replacement, object)
```

- Sub replaces first occurrence only, qsub does them all.

```
cat( "...", file=...)
```

- Outputs text to a file, or prints it on screen if file=""
 - cat requires "\n" to be given for new lines ... try ...

```
cat("Hello World!") ; cat("Hello World!",sep="\n") ; cat("Hello
World!",sep="\n",file="world.txt")
```

- cat is extremely useful for writing scripts or generating reports on-the-fly

Error reporting for extending scripts & user functions

- Your code should report errors if inconsistency is detected.

```
stop(...)
```

- Stops execution of a function and reports a custom error message

```
is.family(...)
```

- Functions that can be used to test for a variety of conditions place them inside **if** structures to check that all is well

```
if ( !is.numeric(x) ){ stop ("Non numeric value entered. Cannot
  continue.") }
```

If the object *x* is non numeric (e.g. Text has been entered when numbers were required), then stop execution and report message

The `is.family`

array	is.language	is.primitive
atomic	is.leaf	is.RF
baseNamespace	is.loaded	is.R
is.call	is.logical	is.raw
is.character	is.null	is.recursive
is.class	is.mts	is.relistable
is.data.frame	is.na	is.symbol
is.classification	is.na.x	is4
is.data.frame	is.na.data.frame	is4e6
is.data.frame	is.na.x.default	is4e6Method
is.debugged	is.na.x.factor	is5e6
is.double	is.na	is5e6Method
is.element	isNamespace	isStepfun
is.environment	is.environment	is.symbol
is.environment	is.na.POSIXt	isSymfun
is.expression	is.null	isSymmetric.matrix
is.factor	is.numeric	is.table
is.finite	is.numeric.date	isTRUE
is.integer	is.na.POSIXt	isTRUE_
is.numeric	is.numeric.version	is.UNKNOWN
is.identicalSymbol	is.numeric.version	is.UNKNOWN
is.identicalSymbol	is.numeric.version	is.UNKNOWN
is.incomplete	is.ordered	is.vector
is.integer	is.ordered	isVirtualClass
is.integer	is.package.version	isVCLClass

Temperature conversion exercise

User functions

- Centigrade to Fahrenheit conversion is given by

- $F = \frac{9}{5} C + 32$

- Write a function that converts between temperatures.

- The function will need two named arguments

- temperature (t) is numeric

- units (**unit**) is character
- They will need default values, e.g $t=0$, $unit="c"$

- The function should report an appropriate error if inappropriate values are given

```
if( !is.numeric(t) ) { .... }
```

```
if( !(unit %in% c("c","f")) ){...}
```

- The function should print out the temperature in F if given in C, and vice versa

Functions with named arguments are defined with the following syntax

```
myFunc<-function(arg=defaultValue, )
```

- Why not add a third scale?

$$K = C + 273.15$$

Example code:
12 convTemp.R

Building the solution

- It is difficult to write large chunks of code, instead start with something that works and build upon it
- E.g. to solve the temperature conversion exercise:
 - start with the function `powXplusX` (from some slides ago)
 - modify the argument names
 - delete the old code, for now just print out the input arguments
 - save the function file, load it into R and try it out
 - now add the two lines for input checking from the previous slide
 - try it out, see if passing a character for temperature gives the expected error
 - now try to convert C into F and print out the result
 - when that works, add the conversion from F to C
- If you get stuck, call us to help you !

Temperature conversion script

```
convTemp<-function(t=0,unit="c"){ # convTemp is defined as a new user function requiring two
arguments, t and unit, the default values are 0 and "c", respectively.

  if( !is.numeric(t) ){
    stop("Non numeric temperature entered") # Exception error if character given for
temperature
  }

  if( !(unit %in% c("c","f","k")) ){
    stop("Unrecognized temperature unit. \n Enter either (c)entigrade, (f)ahrenheit
or (k)elvin") # Exception error if unrecognized units entered
  }

  # Conversion for centigrade
  if(unit=="c"){
    fTemp <- 9/5 * t + 32
    kTemp <- t + 273.15
    output <- paste(t,"C is: \n",fTemp,"F \n",kTemp,"K \n")
    cat(output)
  }

  # Conversion for Fahrenheit
  if(unit=="f"){
    cTemp <- 5/9 * (t-32)
    kTemp <- cTemp + 273.15
    output <- paste(t,"F is: \n",cTemp,"C \n",kTemp,"K \n")
    cat(output)
  }

  # Conversion for Kelvin
  if(unit=="k"){
    cTemp <- t - 273.15
    fTemp <- 9/5 * cTemp + 32
    output <- paste(t,"K is: \n",cTemp,"C \n",fTemp,"F \n")
    cat(output)
  }
}
```

Units must be entered in quotes, as it's a character object

Example code:
12_convTemp.R
