CT5102: Programming for Data Analytics

Lecture 2: Functions in R

Dr. Jim Duggan,
School of Engineering & Informatics
National University of Ireland Galway.
https://github.com/JimDuggan/PDAR
https://twitter.com/_jimduggan

Functions

- A function is a group of instructions that:
  - takes input,
  - uses the input to compute other value, and
  - returns a result (Matloff 2009).
- Functions are a fundamental building block of R (Wickham 2015)
- 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 { }
- Last evaluation is returned

Example

```
> f <- function(x){x^2}
>
> f(4)
[1] 16
```


```
> f(1:10)
 [1]   1   4   9  16  25  36  49  64  81 100
```

Functions in R

- All R functions have 3 parts:
  - The body(), the code inside the function
  - The formals(), list of arguments which controls how you can call the function
  - The environment(), the "map" of the location of the function's variables

```
>
> f
function(x){x^2}
> body(f)
{
    x^2
}
> formals(f)
$x
```

```
> environment(f)
<environment: R_GlobalEnv>
```

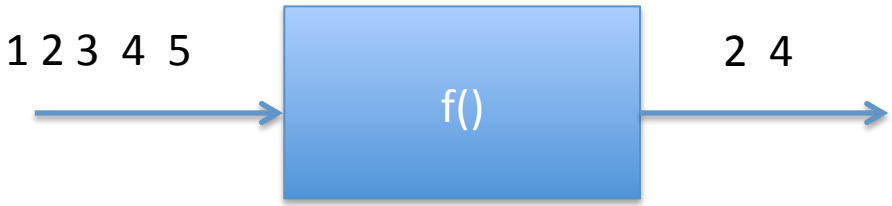
Exception… Primitive Functions

- Only found in the base package (C functions)
- Example: sum()
- Check:
  - is.function()
  - is.primitive()

```
>
> f <- function(x){x^2}
>
> is.function(f)
[1] TRUE
>
> is.primitive(f)
[1] FALSE
>
> is.function(sum)
[1] TRUE
>
> is.primitive(sum)
[1] TRUE
```

Challenge 2.1

- Write an R function that filters a vector to return all even numbers



Lexical Scoping

- Scoping is the set of rules that govern how R looks up the value of a symbol (Wickham 2015)
- Four principles behind R's implementation of lexical scoping
  - name masking
  - functions vs. variables
  - a fresh start
  - dynamic lookup

(1) Name masking – predict the output

```
f <- function(){
  x <- 1
  y <- 2
  c(x,y)
}

f()
```

```
> f()
[1] 1 2
```

Looking for variables…

- If a name isn't defined inside a function, R will look one level up.

```
x <- 2
g <- function(){
  y <- 1
  c(x,y)
}

g()
```

```
>
> g()
[1] 2 1
```

Similar rules apply for nested functions

```
x <- 1
h <- function(){
  y <- 2
  i <- function(){
    z <- 3
    c(x,y,z)
  }
  i()
}

h()
```

```
> h()
[1] 1 2 3
```

Challenge 2.2

- Predict the output of the following call h()
- Use a nested diagram to explain the solution

```
x <- 1; y <- 2; z <- 3
h <- function(){
  x <- 10
  z <- 30
  i <- function(){
    x <- 100
    z <- 300
    j<- function(){
      x <- 1
      c(x,y,z)
    }# end of j
    j()
  } # end of i
  i()
}# end of h
```

(2) Functions vs Variables

- Finding functions works in the same way as finding variables

```
l <- function(x){x+1}

m<-function(){
  l<-function(x){x*2}
  l(10)
}

m()
```

```
> m()
[1] 20
>
```

(3) Values between invocations…

- Every time a function is called, a new environment is created to host the execution.

```
j<- function(){
  if(!exists("a")){
    a<-1
  } else{
    a<-a+1
  }

  print(a)

}
```

```
>
> j()
[1] 1
>
> a<-100
>
> j()
[1] 101
```

Function Arguments

- It is useful to distinguish between *formal arguments* and the *actual arguments*
  - **Formal arguments** are the property of the function
  - **Actual arguments** can vary each time the function is called.
- When calling functions, arguments can be specified by
  - Complete name
  - Partial name
  - Position

```
f <- function(abcdef, bcde1, bcde2){
  c(a=abcdef, b1=bcde1, b2=bcde2)
}
```

```
f(1,2,3)

f(2,3,abcdef = 1)

f(2,3,a = 1)

f(2,3,b = 1)
```

```
> f(1,2,3)
 a b1 b2
 1  2  3
>
> f(2,3,abcdef = 1)
 a b1 b2
 1  2  3
>
> f(2,3,a = 1)
 a b1 b2
 1  2  3
>
> f(2,3,b = 1)
Error in f(2, 3, b = 1) : 
```

Guidelines (Wickham 2015)

- Use positional mapping for the first one or two arguments (most commonly used)
- Avoid using positional mapping for less commonly used attributes
- Named arguments should always come after unnamed arguments

Default and missing arguments

- Function arguments in R can have default values
- R function arguments are "lazy" – only evaluated if actually used

```
g <- function(a=1, b=1){
  c(a,b)
}
```

```
>
> g()
[1] 1 1
> g(1)
[1] 1 1
> g(2,4)
[1] 2 4
```

Status of an argument

- You can determine if an argument was supplied by using the missing() function

```
h <- function(a=1, b=1){
  c(missing(a),missing(b))
}
```

```
>
> h()
[1] TRUE TRUE
>
> h(1)
[1] FALSE  TRUE
>
> h(1,1)
[1] FALSE FALSE
>
```

Creating robust functions

- Defensive programming "the art of making code fail in a well-defined manner even when something unexpected occurs" (Wickham 2015)
- Key principle:
  - Fail fast
  - As soon as something is wrong, signal an error

Examples

```r
my_min <- function(v){
  if(!is.numeric(v))
    stop("Error, type should be numeric")
  min(v)
}
```

```
> my_min("ABC")
Error in my_min("ABC") : Error, type should be numeric
>
> my_min(c(T,F))
Error in my_min(c(T, F)) : Error, type should be numeric
>
> my_min(c(T,F,1))
[1] 0
>
> my_min(300:400)
[1] 300
```

Fail Fast Principle (Wickham 2015)

- Be strict about what you accept
- Avoid functions that use non-standard evaluation (subset, transform)
- Avoid functions that return different types depending on their input. [and sapply().

```r
my_min <- function(v){
  if(!is.numeric(v))
    stop("Error, type should be numeric")
  min(v)
}
```

Return values

- The last expression evaluated in a function becomes the return value, the result of invoking the function
- Generally good style to reserve the use of an explicit **return()** when returning early

```
f<- function(x){
  if( x < 20){
    0
  }else {
    10
  }
}

g<- function(x){
  if( x < 20){
    return(0)
  }else {
    return(10)
  }
}
```

Additional Function Topics

- Function design
  - Infix functions (functions as operators)
  - Can override operators (not recommended!)
  - Replacement Functions
  - Pure functions
- Other topics
  - ... argument
  - do.call()
  - on.exit()

```
> 12+4
[1] 16
>
> '+'<-function(a,b){a*b}
>
> 12+4
[1] 48
>

Restarting R session...

> 12+4
[1] 16
```

Missing Values

- Specified with NA, a logical vector of length 1
- NA (Not Available)
- Useful function: is.na()

```
>
> v<-c(T,F,NA)
>
> v
[1]  TRUE FALSE    NA
>
> typeof(v)
[1] "logical"
>
> is.na(v)
[1] FALSE FALSE  TRUE
>
> v[!is.na(v)]
[1]  TRUE FALSE
```

Challenge 2.2

Write a function that takes in a vector and returns a vector with no duplicates. Make use of the R function duplicated()

duplicated {base} R Documentation

Determine Duplicate Elements

Description

`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.

`anyDuplicated(.)` is a "generalized" more efficient shortcut for `any(duplicated(.))`.

Usage

```
duplicated(x, incomparables = FALSE, ...)
```

References

- Wickham, H. 2015. Advanced R. Taylor & Francis