Basic For-Loop Statement (Lander Chapter 10)

- Loops can be used to follow repetitive operations according to a counter variable taking a particular value out of a set of values. Its syntax is

```
for (counter in setOfValues) {
        lines of statements using counter
      }
```

- If the counter variable is **`i`** and the set of integer values is **`1:n`**, i.e., the set of values $\{1,2,\ldots,n\}$, then **`i`** takes successively the values from 1 to n.
- The counter variable can be used to index values in an array, e.g., **`vec[i]`**.
- See sample R script **`forLoopExamples.r`** .

Basic Logical Statement (Lander Chapter 9)

- The execution of specific lines of operations can be controlled by if-statements. The syntax is

```
if(logical expression) {if expression == T then execute statements}
```

- Logical values can be generated by functions, e.g., **`is.numeric( )`** , or expressions

Expression	Outcome
`(variable == value)`	*TRUE* if the variable has the value
`!logical`	Negation: changes logical *TRUE* => *FALSE* and *FALSE* => *TRUE*
`<`, `>`, `<=`, `>=`, and `!=`	*Less than*, *greater than*, *less equal than*, *greater equal than* and *not equal than*
`&` and `|`	Logical *AND* and *OR* comparing a right-hand and a left-hand logical value

- See sample R script **`ifLogicalExamples.r`** .

Writing Functions (Lander Chapter 8)

Basic Definition of a Function

- A function is defined by

```
fnName <- function(list of arguments) {
  lines of operations
  return(result)
}
```

- The lines of operations within a function are enclosed by `{ … }`
- Once the command `return(output)` is reached within the function's lines of operations the function terminates and returns the output data structure.
- A function is call by `>result <- fnName(list of arguments)`

Prototype Code of the Product of Valid Input Values

- Input: Numeric vector *(are the input values valid?)*
- Output: Product and number of valid factors *(how is the result initialized?)*
- Input check:
  - Is input numeric *(which logical function is used and why perform this test first?)*
  - Remove missing values *(how is it done by indexing vector components?)*
  - At least two factors in input *(which function gives the length of a vector?)*

```
myProdFct <- function(x){
  ###########################################
  ## Input: numeric vector x              ##
  ## Output: product and # of valid data   ##
  ###########################################
```

```
  if (!is.numeric(x)) stop("Input vector not numeric")
  x <- x[!is.na(x)]      # remove any missing values
  n <- length(x)
  if (n < 2) stop("At least 2 factors are needed")

  x.prod <- 1            # initialize with neutral factor
  for (i in 1:n) {
    x.prod <- x.prod * x[i]
  } #end::for
  return( c(x.prod, n) )  # returns a vector
} # end:: myProdFct

v <- c(1,3,-1,NA,-4)
myProdFct(v)           # Test with clean data
myProdFct(v)[1]        # Just the product
myProdFct(v)[2]        # Number of valid factors

myProdFct(v[1])        # Test insufficient number of factors

myLet <- c("q",3,4,1)  # Incorrect data input type
myProdFct(myLet)

prod(v)                # internal R function "prod"
prod(v, na.rm=TRUE)
```

Working with Nominal and Ordinal Scaled Data in R

Review of Nominal and Ordinal Scaled Variables

- Measurements on a ***nominal scale*** just express whether a particular observation has a specific property (attribute or characteristic) or not.
  Usually the number of distinct properties is small.
  There is no natural order among the different categories.

  Examples:

  - color of car => “red”, “yellow”, “blue”, “black”, “neither”
  - city of primary residence => “Plano”, “Dallas”, “Fort Worth”, “neither”

  Note: the “neither” category is required because some observations may not belong to specific labeled categories. This may be different from the missing observations value **`NA`**.

  Categories may be merged together: (“red”, “yellow”) => “light”, (“blue”, “black”) => “dark”, “neither”
  This process is known as ***recoding*** factors.

- Measurements on an ***ordinal scale*** have a particular natural order but distances among the specific representations are not defined. The representation cannot be reordered.

  Examples:

  - Income => “low”< ”middle”< ”high”
  - Income => “[14,40]” < “(40,70]” < “(70,96]” classified into income brackets ($1000)

  Note: here the categories satisfy a particular order.

Organization of the nominal and ordinal scaled observations

- List of individual observations with explicit labels for the categories.

Observation	Factor
1	“female”
2	“female”
⋮	⋮
n	“male

This is usually the format in which the data arrive on the analyst’s desk. The **`data.frame`** functions automatically convert the labeled variable into a **`factor`**.

The ***lexicographically*** lowest category label receives the value 1 and so forth.

If one wants the labels to appear in a natural order, then one would need to convert the factor into an ***ordered factor***.

- List of individual observations with ***numerical encoding*** of the categories and ***lookup table*** for the codes.

Data-Frame	
Observation	*Factor*
1	1
2	1
⋮	⋮
n	2

Lookup Table	
Code	*Label*
1	"female"
2	"male"

This is R’s representation of a factor. However, R automatically shows the label of each category rather than its numerical placeholder. The labels, however, are _not_ strings and therefore _not_ enclosed by quotation marks.

Key functions related to factors

See online help for the use of these functions.

- **`factor()`**: generates a factor based on the unique set of representation of a variable
- **`ordered()`**: converts the factor levels into an ordinal scaled variable.
- **`table()`**: converts a list of individual observations into a table of counts based on a factor levels. With more than one factor as input it generates a cross-tabulation.
- **`xtabs()`**: converts an aggregated list into a table.
- **`as.matrix()`**: changes a bivariate table into a matrix where the rows have names of their associated factor levels and vice versa for the columns.
- **`as.numeric()`**: strips the labels from a factor and just leaves the numerical code of the categories.
- **`to.data.frame()`**: converts any string variable into a factor
- **`cut()`**: converts a metric variable into a factor or an ordered factor by classifying the observations according to specific value ranges.
- **`prop.table()`**: calculates row or column percentages of a table
- **`addmargins()`**: calculate row or column sum and adds these to the margins of the table

Numerical Precision of Floating Point Numbers

- For a concise review see Chapter 2: *Sources of Inaccuracy in Statistical Computation* in Altman, Micah, Jeff Gill, and Michael P. McDonald (**2004**) *Numerical Issues in Statistical Computing for the Social Scientist*. John Wiley
- It is important to understand the digital representation of real numbers, which are defined as fractions of integers, and irrational numbers (numbers with an infinite number of digest such as $\pi = 3.141593 \ldots$ or simply $1/3 = 0.33333 \ldots = 0.\overline{3}$, see https://en.wikipedia.org/wiki/Irrational_number) and to learn about
  a. the ***feasible value range*** of digital number systems,
  b. the ***smallest representable difference*** between numbers,
  c. problems of combining numbers on ***different numerical scales***, and
  d. sources for ***rounding errors***.
- Each elementary memory cell of a digital computer can only store ***two states***, e.g., $\{\text{off,on}\}$ or $\{0,1\}$. These elementary memory cells are called ***bits***.
- To simply the discussion assume that a computer system only uses 4 bits[1] to store numbers. These four bits are (b_1, b_2, b_3, b_4) with $b_i \in \{0,1\}$ and the bit b_4 being associated with the ***exponent*** of numbers on a ***floating scale***.
- In addition to floating point numbers also integer numbers are used, which use the last bit b_4 as part of the integer number.
- Representation of integer and floating point numbers:
  - For ***integer numbers*** the 4 bits are mapped to an integer $i(b_1, b_2, b_3, b_4)$ by

$$\begin{aligned} i(b_1, b_2, b_3, b_4) &= b_4 \cdot 8 + b_3 \cdot 4 + b_2 \cdot 2 + b_1 \cdot 1 \\ &= b_4 \cdot 2^3 + b_3 \cdot 2^2 + b_2 \cdot 2^1 + b_1 \cdot 2^0 \end{aligned}$$

[1] Note the IEEE double precession number system uses 64 bits to represent numbers: one for the sign, 11 for the exponent, and 52 for the fraction.

- For ***floating point*** numbers the 4 bits are mapped to a decimal number $d(b_1, b_2, b_3, b_4)$ by

$$\begin{aligned} d(b_1, b_2, b_3, b_4) &= (b_3 \cdot 4 + b_2 \cdot 2 + b_1 \cdot 1) \times 2^{b_4} \\ &= (b_3 \cdot 2^2 + b_2 \cdot 2^1 + b_1 \cdot 2^0) \times 2^{b_4} \end{aligned}$$

- For the $16 = 2^4$ permutations of the 2 states of these 4 bits the following integer and floating point numbers can be generated:

b_1	b_2	b_3	b_4	$i(b_1, b_2, b_3, b_0)$	$d(b_1, b_2, b_3, b_4)$
0	0	0	0	0	0
1	0	0	0	1	1
0	1	0	0	2	2
1	1	0	0	3	3
0	0	1	0	4	4
1	0	1	0	5	5
0	1	1	0	6	6
1	1	1	0	7	7
0	0	0	1	8	***0***
1	0	0	1	9	***2***
0	1	0	1	10	***4***
1	1	0	1	11	***6***
0	0	1	1	12	***8***
1	0	1	1	13	***10***
0	1	1	1	14	***12***
1	1	1	1	15	***14***

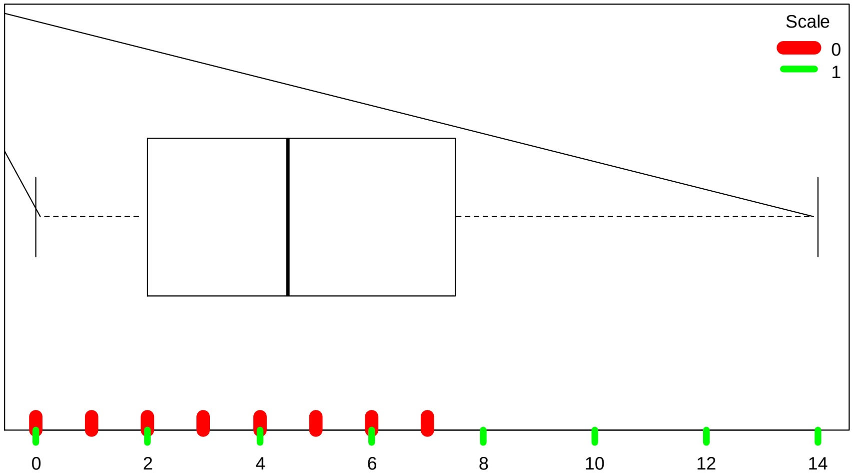

Figure 1: Floating point numbers on different scales

- For integers the ***difference*** between two adjacent numbers is ***fixed***.
- R currently uses 32-bit integers ranging from −2,147,483,647 through +2,147,483,647.
- In contrast, for floating point numbers the ***difference*** depends on the ***used scale*** (the status of the exponential bit b_4)
- Depending on the status of the exponential bit b_0 a few properties of floating point numbers become apparent:
  a. Both scales ***overlap*** to some degree. This redundancy allows representing identical numbers on different scales.
     Representing a number on the ***lowest possible scale*** is called ***normalization***.
  b. ***Shifting*** the sequence of bits b_1, b_2, b_3 of a number one bit ***to the right effectively doubles*** the number. If the right most bit b_3 is $b_3 = 1$ an ***overflow*** occurs and the scale bit b_4 changes from 0 to 1.

c. The scale associated with $b_4 = 1$ represents a wider spaced sequence of numbers, which results in a lower ***numerical resolution***.
d. Numbers ***outside the available value range*** 0 to 14 are interpreted as ***infinite*** numbers.
e. To ***map*** a number associated with the scale $b_4 = 0$ into the scale associated with $b_0 = 1$ it may need to ***rounded*** to its next representable number in the $b_4 = 1$ scale.
   For example: the number 5 in the $b_4 = 0$ scale needs to be rounded to either numbers 4 or 6 in the $b_4 = 1$ scale.
f. To ***perform operations*** with two numbers on different scales the smaller number needs to be first mapped into the higher scale.
   For example: If we want to add the number 3 in the $b_4 = 0$ scale to the number 8 in the $b_4 = 1$ scale, the value 3 first needs to be mapped into the $b_4 = 1$ scale, which give either 2 or 4.
   Therefore, depending on the applied rounding rule the result will either be 10 or 12.
   Consequence: operations between different scales may induce rounding errors.

- 64-bit double precision numbers have a value range of approximately -10^{308} through +10^{308}. Any value outside this range is infinity.
- See sample script **`ExploreNumericalStability.r`**.
- For more on the ANSI/IEEE 754-1985 double precision floating points numbers see https://en.wikipedia.org/wiki/Double_precision_floating-point_format
- Higher precision number systems are available. See for instance https://en.wikipedia.org/wiki/Arbitrary-precision_arithmetic with possible applications in:
  - Landing a fridge sized probe (Philae) after 10 years of flight on a fast moving comet 67P half a billion kilometers away from earth needs to be done with more precise digital number systems. See https://en.wikipedia.org/wiki/Rosetta_(spacecraft)

   - To calculate the precise location using GPS signals not only numerical errors need to be accounted for but also other causes of error variations such as Einstein's relativity theories since time on the fast moving satellites is moving slower. See https://en.wikipedia.org/wiki/Error_analysis_for_the_Global_Positioning_System
- Ultimately, the numerical representation of floating point numbers becomes sparser as the exponential factor2^e increases. Operations performed on numbers in different scales will lead to possible rounding errors to lift the lower scale numbers up onto an upper scale.