Functions in R

Writing Functions

- **R** language allows the user to create objects of mode ***function***.

Keyword **function** Arguments

- A **function** is defined by an assignment of the form:

```
> fname <- function(arg_1, arg_2, …){
        . . expression (body). .}
```

Body

```
The body is an R expression,(usually a grouped
expression), that uses the arguments to calculate
a value which is the value returned by the
function.
```

- A **call** to the function usually takes the form: > **`fname(expr_1, expr_2,...)`** and may occur anywhere a **function call** is legitimate.

Writing Functions

- A ***function*** is an object in **R** that takes some *input objects* (the **arguments** of the function) and returns an *output object*.
- R functions only return the last ‘thing’ done.
- All work in **R** is done by functions.

Example: Two-Sample t-Statistic

- Consider a function to calculate the two sample t-statistic, showing “all the steps”.
- This function may be defined as follows:

```
> twosam <- function(y1, y2) {
     n1  <- length(y1); n2  <- length(y2)
     yb1 <- mean(y1);   yb2 <- mean(y2)
     s1  <- var(y1);    s2  <- var(y2)
     s <- ((n1-1)*s1 + (n2-1)*s2)/(n1+n2-2)
     tst <- (yb1 - yb2)/sqrt(s*(1/n1 + 1/n2))
     tst
   }
```

Then we can call this function:

```
> tsat <- twosam(data$male,data$female); tsat
```

Evaluation Environment of a Function

- When a function is invoked, a new ***evaluation frame*** is created.
- Formal arguments are matched with supplied arguments according to the ***rules of argument matching***:
  1) **Exact matching on tags**: For each named supplied arguments the list of formal arguments is searched for an item whose name matches exactly.
  2) **Partial matching on tags**: Each named supplied argument is compared to the remaining formal arguments using partial matching.
  3) **Positional matching**: Any unmatched formal arguments are bound to unnamed supplied arguments, in order. Any . . . argument will take up the remaining arguments, tagged or not.

Named Arguments and Defaults

- If arguments to called functions are given in the `"name=object"` form, they may be given in any order.
- For example, if we define function **`fun1`** as:

```
> fun1 <- function(data, data.frame, graph, limit){
     [function body omitted]
     }

We can then invoke the function in several ways:

> ans <- fun1(d, df, TRUE, 20)
> ans <- fun1(d, df, graph=TRUE, limit=20)
> ans <- fun1(data=d, limit=20, graph=TRUE, data.frame=df)
```

Named Arguments and Defaults

- If arguments are given in commonly appropriate default values, they may be omitted altogether from the call:
- For example, if we define function **`fun1`** as:

```
> fun1 <- function(data, data.frame, graph=TRUE, limit=20){
     [function body omitted]
     }
It could be called as:
> ans <- fun1(d, df)
Which is equivalent to the three previous calls. However:
> ans <- fun1(d, df, limit=10)
changes one of the defaults.
```

Assignments Within Functions

- *Any ordinary assignments done within the function are **local** and temporary and are lost after exit from the function.*
- So the assignment **`X <- qr(X)`** does not affect the value of the argument in the calling function.
- If global and permanent assignments are intended within a function, then either the **superassignment** operator **`<<-`** or the function **`assign()`** can be used.

Scope

- The symbols in the body of a function can be divided into three classes:
  - **Formal parameters** of a function: those occurring in the argument list of the function.
  - **Local variables**: those whose values are determined by the evaluation of expressions in the body of the function.
  - **Free variables**: variables which are not formal parameters or local variables.

Scope Example

- Consider the following function definition:

```
f <- function(x){
    y <- 2*x
     print(x)
     print(y)
     print(z)
    }
```

x is a ***formal*** parameter. All formal arguments provide **bound** symbols in the body.

y is a ***local*** variable

z is a ***free*** (and initially **unbound**) variable

`In this function,` **`x is a formal parameter, y is a local variable`** `and` **`z is a free variable.`**

Lexical Scope

- In **R** the free variable bindings are resolved by first looking in the environment in which the function was created.
- We define a function called **`cube`**:

```
> cube <- function(n){
    sq <- function() n*n
     n*sq()
    }
```

The variable n in the function sq() is not an argument to that function. It is a free variable and the ***scoping rules*** must be used to ascertain the value associated with it.

Optional Arguments

- Function **`charplot()`** with two essential (x and y) and two optional arguments (pc and co):

```
> charplot <- function(x,y,pc=16,co="red"){
  plot(y~x,pch=pc,col=co)}
To execute, you only need to provide x and y:
> charplot(1:10,1:10)
To get a different plotting symbol:
> charplot(1:10,1:10,17)
For navy-colored circles:
> charplot(1:10,1:10,co="navy")
To change both plotting symbol and color:
> charplot(1:10,1:10,15,"green")
```

Red solid circles (pch=16)

Red solid triangles (pch=17)

Navy circles

Green squares

Optional Arguments

- Function **`charplot()`** with two essential (x and y) and two optional arguments (pc and co):

```
> charplot <- function(x,y,pc=16,co="red"){
  plot(y~x,pch=pc,col=co)}
```

`Reversing arguments does not work:`

```
> charplot(1:10,1:10, "green",15)
```

`Order unimportant if specify both variable names:`

```
> charplot(1:10,1:10,co="green",pc=15)
```

Solid green squares again

Variable Numbers of Arguments (. . .)

- Calculates means and variances of any number of vectors:

```
> many.means <- function(...){
    data <- list(...)
    n <- length(data)
    means <- numeric(n)
    vars <- numeric(n)
    for (i in 1:n){
      means[i] <- mean(data[[i]])
      vars[i] <- var(data[[i]])
    }
    print(means)
    print(vars)
    invisible(NULL)
  }
```

Creates a list object ‘data;’
Length is number of vectors

‘means’, ‘vars’ have same # elements as there are vectors

Lazy Evaluation of Function Arguments

- Let's try it out ! :

```
> x <- rnorm(100)
> y <- rnorm(200)
> z <- rnorm(300)
> many.means(x,y,z)
[1]  -0.039181830  0.003613744  0.050997841
[1]      1.146587      0.989700      0.999505
```

Returning Values From a Function

- Example of a function returning a single value:

```
> parmax <- function (a,b){
  c <- pmax(a,b)
  median(c)}
TRY IT OUT:
x <- c(1,9,2,8,3,7)
y <- c(9,2,8,3,7,2)
parmax(x,y)
Unassigned last line median(c) returns a value:
[1] 8
```

- If you want to return two or more variables from a function you should use **`return()`** with a list containing the variables to be returned (next slide).

Returning Values From a Function

- Multiple returns example:

```
> parboth <- function (a,b){
   c <- pmax(a,b)
   d <- pmin(a,b)
   answer <- list(median(c),median(d))
   names(answer)[[1]] <- "median of the par maxima"
   names(answer)[[2]] <- "median of the par minima"
   return(answer)}
```

Example using the same data as before:

```
> parboth(x,y)
$'median of the par maxima'
[1] 8
$'median of the par minima'
[1] 2
```

Anonymous Functions

- **Anonymous** functions in **R** are *unnamed* functions:

```
> (function(x,y){z <- 2*x^2 + y^2; x+y+z})(0:7,1)
[1] 2 5 12 23 38 57 80 107
```

Notice the use of the semicolon to separate the first and second lines in the body of the function.

Flexible Handling of Arguments to Functions

- Want a function to work with either one **or** two arguments:

```
> plotx2 <- function(x,y=z^2){
      z <- 1:x
      plot(z,y,type="l")} # type is lower case l
```

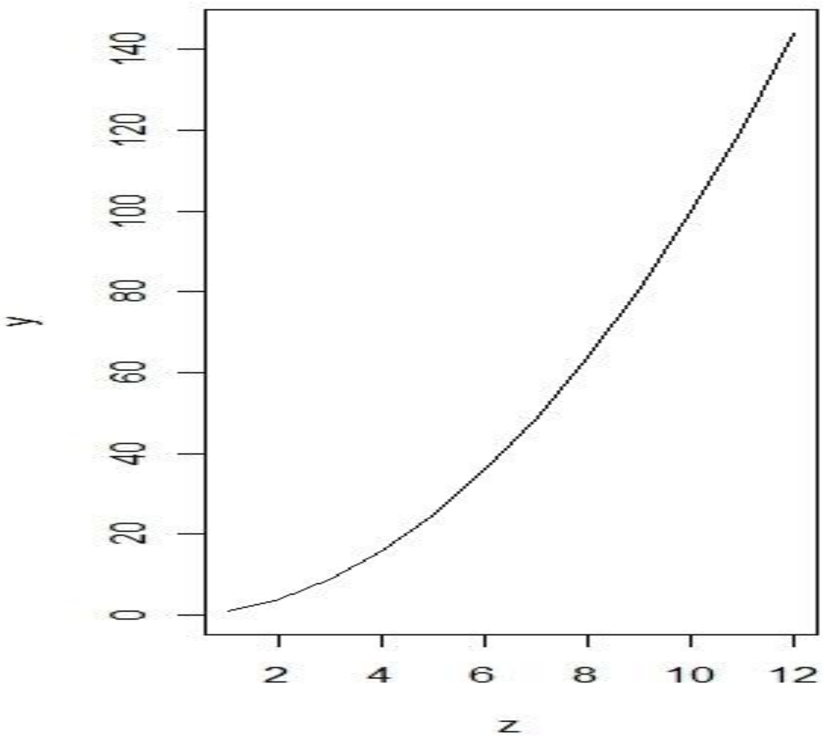
- In many other languages, the ***first line would fail*** because z is not defined at this point.
  - But **R** does not evaluate an expression until the body of the function actually calls for it to be evaluated, which is never in the case where *y* is supplied as the second argument.
  - **One argument:** Get graph of z^2 against z.
  - **Two arguments:** Get graph of y against z.

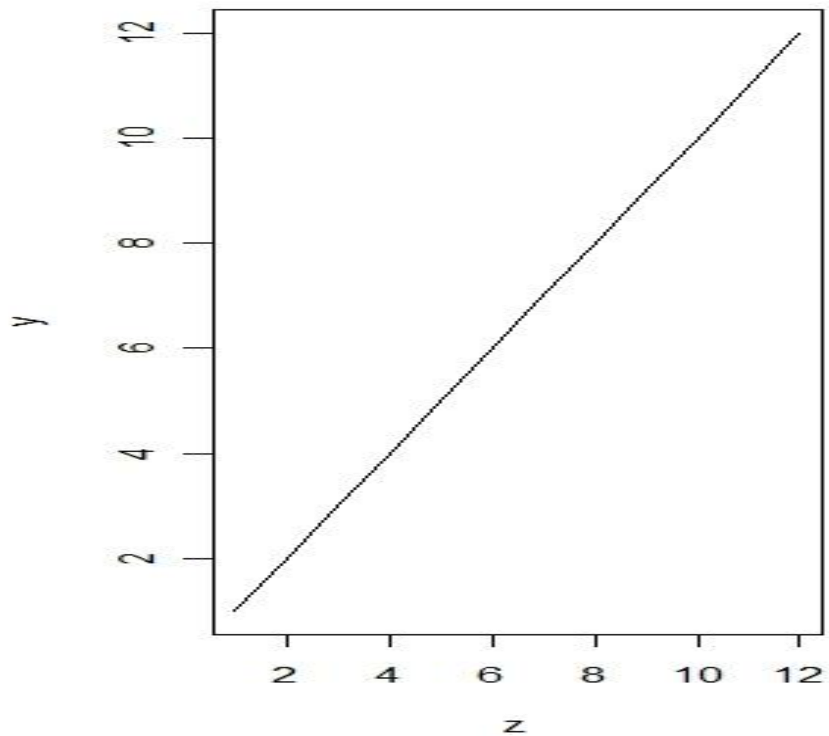
Flexible Handling of Arguments to Functions

- Call function **`plotx2()`**:

```
> par(mfrow=c(1,2))
> plotx2(12)
> plotx2(12,1:12)
```

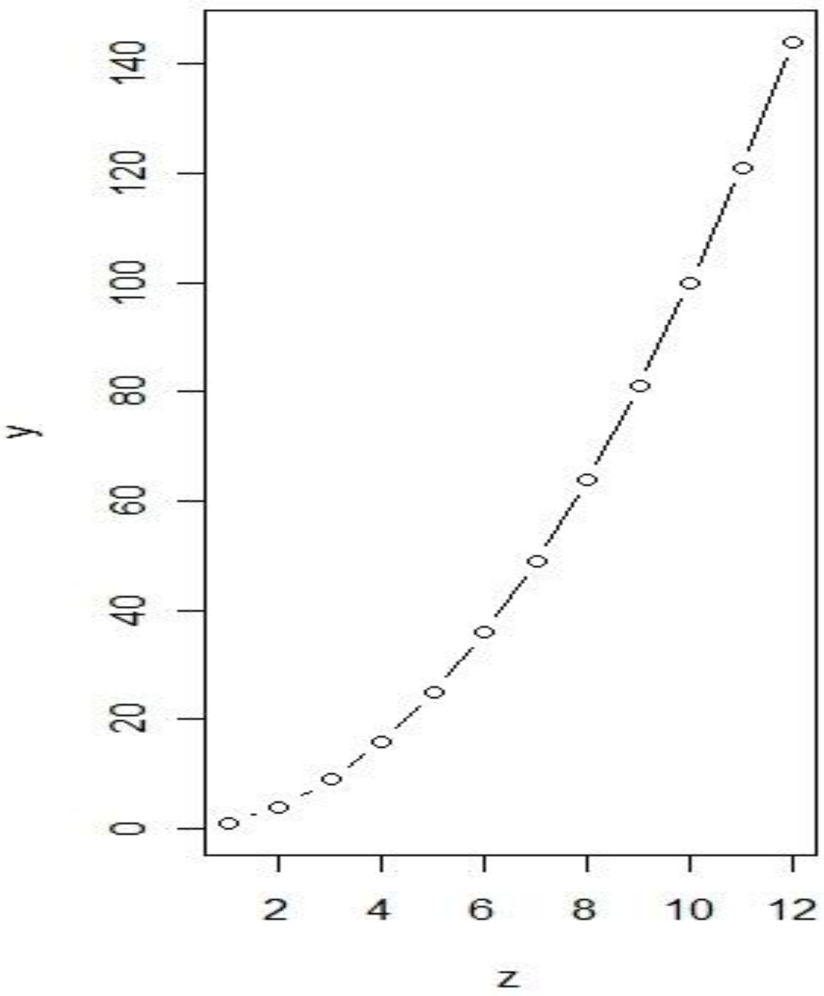
plot(z,y,type="l")

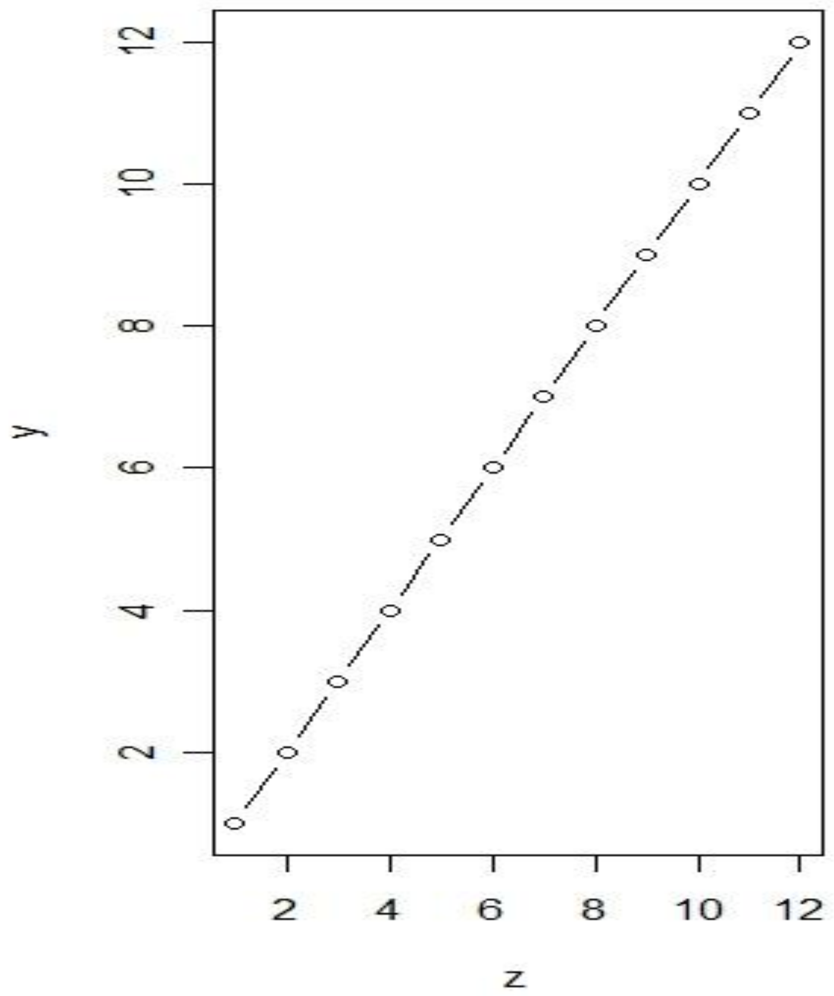




Flexible Handling of Arguments to Functions

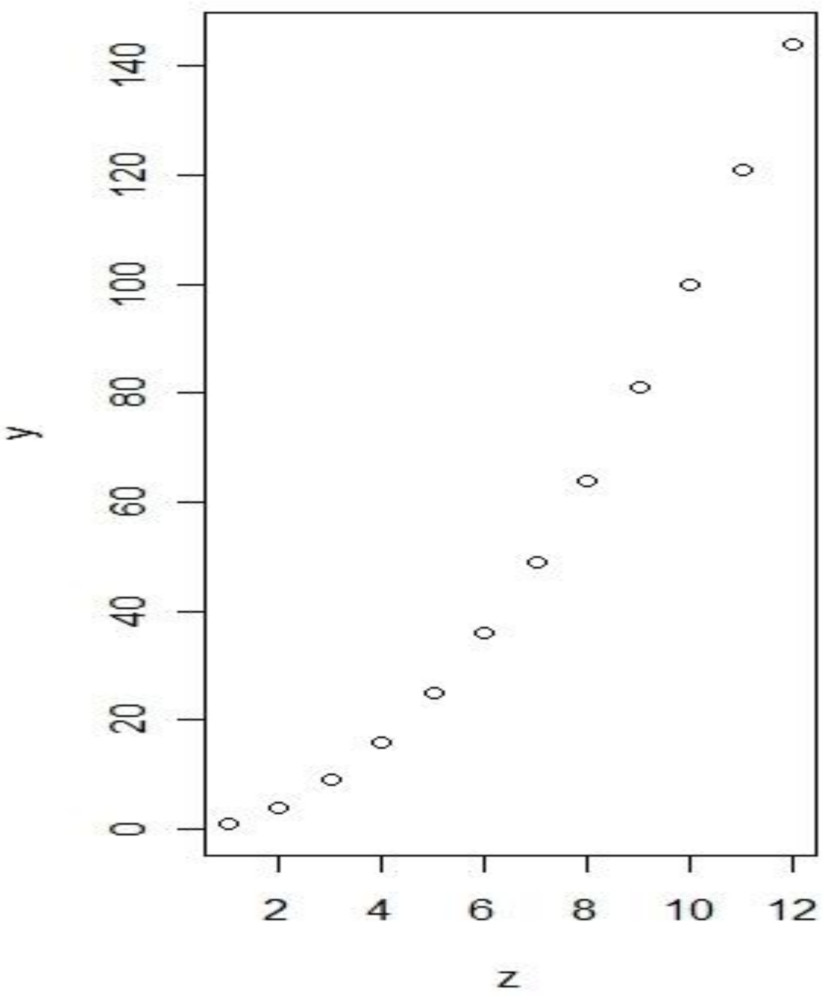
plot(z,y,type="b")

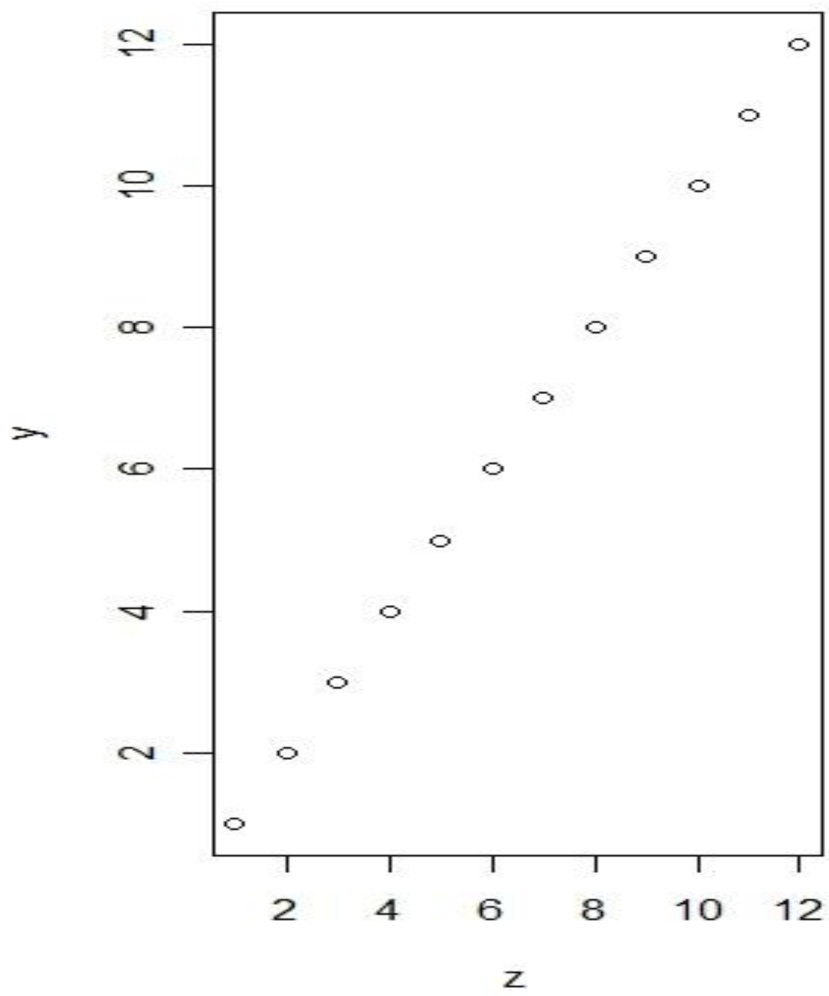




Flexible Handling of Arguments to Functions

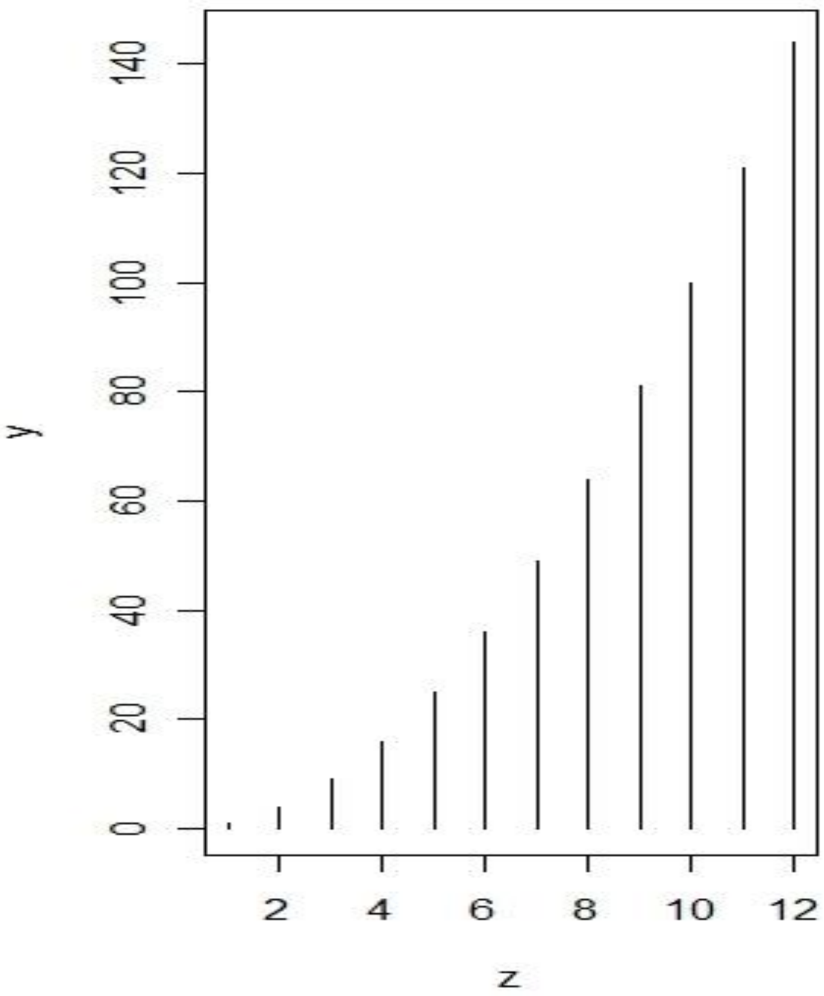
plot(z,y,type="p")

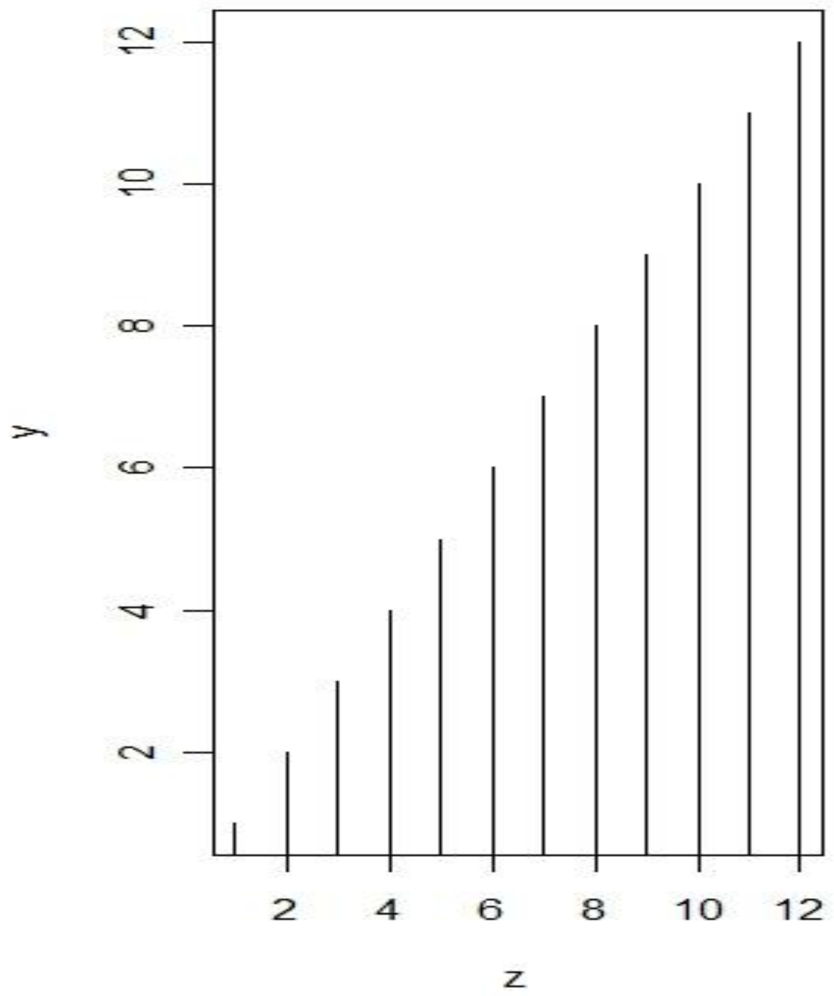




Flexible Handling of Arguments to Functions

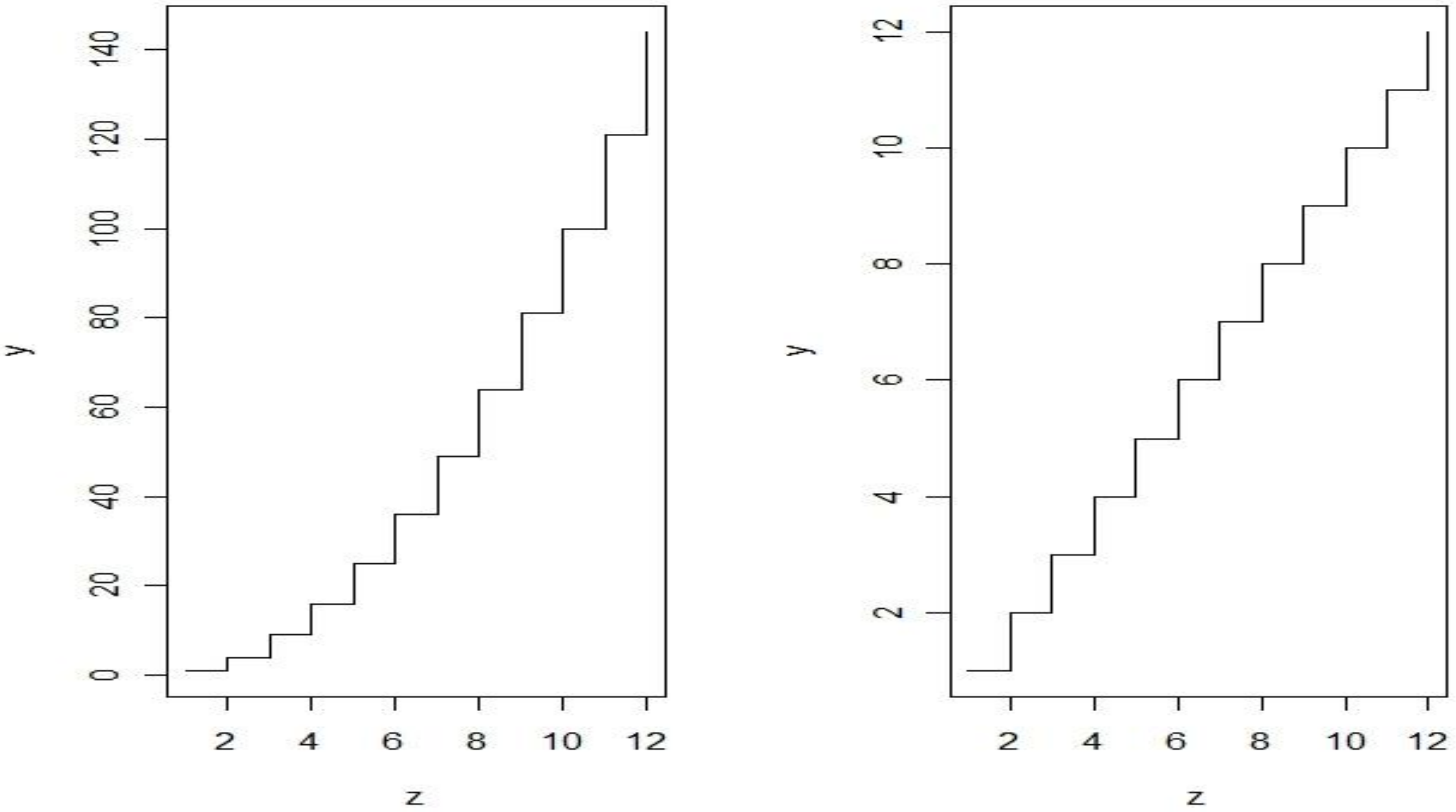
plot(z,y,type="h")





Flexible Handling of Arguments to Functions

plot(z,y,type="s")



Frequently-Used Functions in R

The `ifelse()` function

- Enables you to do one thing if condition is **true** and another if it is **false**:

```
Replace y with negative or positive values:
> z <- ifelse(y < 0,-1,+1)
Convert Area into new, two-level factor:
> data <- read.table("c:\\temp\\worms.txt",header=T)
> attach(data)
> ifelse(Area > median(Area), "big","small")
[1] "big"   "big" "small" "small" . . .
[10] "small" "small"   "big"   "big" . . .
[19] "small" "small"
```

The `ifelse()` function

- Another use of **`ifelse()`** is to override **R's** natural inclinations:

```
Log of zero in R is -Inf:
> y <- log (rpois(20,1.5)); y
[1] 0.0000000 1.0986123 1.0986123
[8] . . .             -Inf       -Inf
[15]  . .             -Inf       -Inf
We want -Inf to be represented with NA:
> ifelse(y < 0, NA,y)
[1] 0.0000000 1.0986123 1.0986123
[8] . . .               NA         NA
[15]  . .               NA         NA
```

The `switch()` function

- Is useful when you need a function to do different things is different circumstances:
- For example, if we define function **`central`** to calculate any one of four different measures of central tendency (arithmetic, geometric or harmonic mean; or median):

```
> central <- function(y,measure){
     switch(measure,
       Mean = mean(y),
       Geometric = exp(mean(log(y))),
       Harmonic = 1/mean(1/y),
       Median = median(y),
     stop("Measure not included"))}
> central(rnorm(100,10,2), "Harmonic")
[1] 9.554712
> central(rnorm(100,10,2),4)
[1] 10.46240
```

The `sample()` Function

- Shuffles the contents of a vector into a random sequence.

```
> y <- c(8,3,5,7,6,6,8,9,2,3,9,4,10,4,11)
Here are two samples of y:
> sample(y)
[1] 8 8 9 9 2 10 6 7 3 11 5 4 6 3 4
> sample(y)
[1] 9 3 9 8 8 6 5 11 4 6 4 7 3 2 10
```

The `sample()` Function

You can specify the size of the sample with the 2nd argument:

```
> sample(y,5)
[1] 9 3 4 2 8
```

Sampling ***with*** replacement;
No 10 and there are three 9's.

```
> sample(y,replace=T)
[1] 9 6 11 2 9 4 6 8 8 4 4 4 3 9 3
```

In this next sample ***with*** replacement there are two 10's and only one 9:

```
> sample(y,replace=T)
[1] 3 7 10 6 8 2 5 11 4 6 3 9 10 7 4
```

apply() Function

- The **apply()** function is used for applying functions to the rows or columns of matrices or dataframes:

```
> ( X <- matrix(1:24,nrow=4))
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    2    6   10   14   18   22
[3,]    3    7   11   15   19   23
[4,]    4    8   12   16   20   24
```

Apply a **sum()** function across the rows or columns:

```
> apply(X,1,sum)
[1] 66 72 78 84
> apply(X,2,sum)
[1] 10 26 42 58 74 90
```

‘1’ is rows; ‘2’ is columns

`apply()` Function

- You can apply functions to the individual elements of a matrix:

Apply `sqrt()` to the rows

```
> apply(X,1,sqrt)
         [,1]     [,2]     [,3]     [,4]
[1,] 1.000000 1.414214 1.732051 2.000000
[2,] 2.236068 2.449490 2.645751 2.828427
[3,] 3.000000 3.162278 3.316625 3.464102
[4,] 3.605551 3.741657 3.872983 4.000000
[5,] 4.123106 4.242641 4.358899 4.472136
[6,] 4.582576 4.690416 4.795832 4.898979
```

`apply()` Function

- You can apply functions to the individual elements of a matrix:

Apply **`sqrt()`** to the columns

```
> apply(X,2,sqrt)
       [,1]   [,2]   [,3]   [,4]   [,5]   [,6]
[1,] 1.0000 2.2361 3.0000 3.6056 4.1231 4.5826
[2,] 1.4142 2.4494 3.1623 3.7417 4.2426 4.6904
[3,] 1.7321 2.6458 3.3166 3.8730 4.3589 4.7958
[4,] 2.0000 2.8284 3.4641 4.0000 4.4721 4.8990
```

`apply()` Function

- You can apply your own function definition within **`apply()`** like this:

```
> apply(X,1,function(x)x^2+x)
     [,1] [,2] [,3] [,4]
[1,]    2    6   12   20
[2,]   30   42   56   72
[3,]   90  110  132  156
[4,]  182  210  240  272
[5,]  306  342  380  420
[6,]  462  506  552  600
```

`sapply()` Function

- To apply a function to a vector, use **`sapply()`**:

```
> sapply(3:7,seq)
[[1]]
[1] 1 2 3
[[2]]
[1] 1 2 3 4
[[3]]
[1] 1 2 3 4 5
[[4]]
[1] 1 2 3 4 5 6
[[5]]
[1] 1 2 3 4 5 6 7
```

`lapply()` Function

```
> a <- c("a","b","c","d")
> b <- c(1,2,3,4,4,3,2,1)
> c <- c(T,T,F)
Create list object with list() function:
> list.object <- list(a,b,c)
> class(list.object)
[1] "list"
To see contents of list:
> list.object
[[1]]
[1] "a" "b" "c" "d"
[[2]]
[1] 1 2 3 4 4 3 2 1
[[3]]
[1] TRUE TRUE FALSE
```

`lapply()` Function

```
> lapply(list.object,length)

[[1]]
[1] 4

[[2]]
[1] 8

[[3]]
[1] 3
```

`length()` returns number of elements comprising each component of the list.

`lapply()` Function

```
> lapply(list.object,class)

[[1]]
[1] "character"

[[2]]
[1] "numeric"

[[3]]
[1] "logical"
```

`class()` returns the class of each component of the list.

`lapply()` Function

```
> lapply(list.object,mean)

[[1]]
[1] NA

[[2]]
[1] 2.5

[[3]]
[1] 0.6666667

Warning message:
argument is not numeric or logical . . .
mean.default(X[[1]],…)
```

What happens when we apply function **`mean()`**?

`tapply()` Function

- The most important function in **R** for generating summary tables is the **`tapply()`** function.
  - Applies a known function (e.g. mean, variance) across specified factor levels to create a table.

```
> data <- read.table("c://temp/Daphnia.txt",header=T)
> attach(data)
> names(data)
[1] "Growth.rate" "Water" "Detergent" "Daphnia"
```

Three factors

```
Want mean growth rates for four detergent brands:
> tapply(Growth.rate,Detergent,mean)
  BrandA    BrandB    BrandC    BrandD
3.884832  4.010044  3.954512  3.558231
```

`tapply()` Function

- Or to tabulate mean growth rates for the two rivers:

```
> tapply(Growth.rate,Water,mean)
    Tyne       Wear
3.685862   4.017948
```

- Or for the three Daphnia clones:

```
> tapply(Growth.rate,Daphnia,mean)
    Clone1      Clone2      Clone3
  2.839875  4.577121  4.138719
```

`tapply()` Function

```
> tapply(Growth.rate,list(Daphnia,Detergent),mean)
         BrandA   BrandB   BrandC   BrandD
Clone1 2.732227 2.929140 3.071335 2.626797
Clone2 3.919002 4.402931 4.772805 5.213745
Clone3 5.003268 4.698062 4.019397 2.834151
```

Can use an ‘anonymous function’ for SEs:

```
> tapply(Growth.rate,list(Daphnia,Detergent),
+ function(x)sqrt(var(x)/length(x)))
          BrandA    BrandB    BrandC    BrandD
Clone1 0.2163448 0.2319320 0.3055929 0.1905771
Clone2 0.4702855 0.3639819 0.5773096 0.5520220
Clone3 0.2688604 0.2683660 0.5395750 0.4260212
```

`tapply()` Function

```
> tapply(Growth.rate,list(Daphnia,Detergent,Water),mean)
,,Tyne
          BrandA    BrandB    BrandC    BrandD
Clone1 2.811265  2.775903  3.287529  2.597192
Clone2 3.307634  4.191188  3.620532  4.105651
Clone3 4.866524  4.766258  4.534902  3.365766
 ,,Wear
          BrandA    BrandB    BrandC    BrandD
Clone1 2.653189  3.082377  2.855142  2.656403
Clone2 4.530371  4.614673  5.925078  6.321838
Clone3 5.140011  4.629867  3.503892  2.302537
```

In cases like this, the function **ftable()** (which stands for ‘flat table’) often produces more pleasing output:

`ftable()` Function

```
> ftable(tapply(Growth.rate,list(Daphnia,Detergent,Water),mean))
                    Tyne      Wear
Clone1 BrandA   2.811265  2.653189
       BrandB   2.775903  3.082377
       BrandC   3.287529  2.855142
       BrandD   2.597192  2.656403

Clone2 BrandA   3.307634  4.530371
       BrandB   4.191188  4.614673
       BrandC   3.620532  5.925078
       BrandD   4.105651  6.321838

Clone3 BrandA   4.866524  5.140011
       BrandB   4.766258  4.629867
       BrandC   4.534902  3.503892
       BrandD   3.365766  2.302537
```

`ftable()` Function

```
> water <- factor(Water,levels=c("Wear","Tyne"),is.ordered(Water))
> ftable(tapply(Growth.rate,list(Daphnia,Detergent,Water),mean))
                    Wear     Tyne
Clone1 BrandA   2.653189 2.811265
       BrandB   3.082377 2.775903
       BrandC   2.855142 3.287529
       BrandD   2.656403 2.597192

Clone2 BrandA   4.530371 3.307634
       BrandB   4.614673 4.191188
       BrandC   5.925078 3.620532
       BrandD   6.321838 4.105651

Clone3 BrandA   5.140011 4.866524
       BrandB   4.629867 4.766258
       BrandC   3.503892 4.534902
       BrandD   2.302537 3.365766
```

`table()` Function

- `table()` is a data summary function that creates a table of counts:

```
> library(DAAG) # use tinting dataframe from DAAG
> table(Sex=tinting$sex, AgeGroup=tinting$agegp)
    AgeGroup
Sex younger older
  f      63    28
  m      28    63
```

- By default, `table()` ignores NAs.

Functions Exercises

Functions Exercises

1. **(a)** Write functions `tmpFn1` and `tmpFn2` such that if `xVec` is the vector $(x_1, x_2, \ldots, x_n)$, then `tmpFn1(xVec)` returns the vector $(x_1, x_2^2, \ldots, x_n^n)$ and `tmpFn2(xVec)` returns the vector $\left(x_1, \frac{x_2^2}{2}, \ldots, \frac{x_n^n}{n}\right)$.

   (b) Now write a function `tmpFn3` which takes 2 arguments `x` and `n` where `x` is a single number and `n` is a strictly positive integer. The function should return the value of

$$1 + \frac{x}{1} + \frac{x^2}{2} + \frac{x^3}{3} + \cdots + \frac{x^n}{n}$$

2. Write a function `tmpFn(xVec)` such that if `xVec` is the vector $\mathbf{x} = (x_1, \ldots, x_n)$ then `tmpFn(xVec)` returns the vector of moving averages:

$$\frac{x_1 + x_2 + x_3}{3}, \quad \frac{x_2 + x_3 + x_4}{3}, \quad \ldots, \quad \frac{x_{n-2} + x_{n-1} + x_n}{3}$$

   Try out your function; for example, try `tmpFn( c(1:5,6:1) )`.