FUNCTION ARGUMENTS

DSC205 Introduction to Advanced Data Science

Marcus Birkenkrahe

April 1, 2023

Arguments

- Watch: "The Argument" by Monty Python (the British comedy group that gave the "Python" programming language its name).

- The English word 'argument' comes from Latin *argumentum* for *make clear, prove, accuse*, and stands for any 'process of reasoning'[1].
- Here: default arguments, missing argument values, and passing extra arguments using ellipses (...)

Lazy argument evaluation

- "Lazy" evaluation refers to the fact that expressions are evaluated only when they're needed by the program (saving time and effort).
- We're going to look at an extended working example: searching through a `list` for `matrix` objects and trying to multiply them with another `matrix` specified as a second argument.

Intermission: matrix multiplication

- In order to multiply two matrices A and B of size m x n and p x q, it must be true that n = p (column length of A == row length of B).

[1]Since I'm writing this lecture on the eve of the visit of the "Pope's astronomer", Br. Guy Consolmagno, a Jesuit and astronomer to Pope Francis (another Jesuit), I am reminded of the reputation of Jesuits for being great at arguing - something you could see in the Q&A session after Gr. Guy's talk on "Astronomy, Religion and the Art of Storytelling". One could argue that any story contains at least one argument (a complete process of reasoning), and that the best arguments are constructed like stories (remember the Freytag curve, which I've also used in my paper on scientific storytelling).

$$\begin{bmatrix} 2 & 5 & 2 \\ 6 & 1 & 4 \end{bmatrix} \cdot \begin{bmatrix} 3 & -3 \\ -1 & 1 \\ 1 & 5 \end{bmatrix}$$

$$= \begin{bmatrix} 2\times 3 + 5\times(-1) + 2\times 1 & 2\times(-3) + 5\times(1) + 2\times 5 \\ 6\times 3 + 1\times(-1) + 4\times 1 & 6\times(-3) + 1\times(1) + 4\times 5 \end{bmatrix}$$

$$= \begin{bmatrix} 3 & 9 \\ 21 & 3 \end{bmatrix}$$

- Unlike `matrix` addition, subtraction and scalar multiplication (with a number), matrix multiplication is not an element-wide operation, and the operator is not `*` but instead `%*%`.
- Example: create two matrices, print them and check their dimension:

```
A <- rbind(c(2,5,2),  # rbind is filled by row
           c(6,1,4))
A
## check dimension
dim(A)
B <- cbind(c(3,-1,1),  # cbind is filled by column
           c(-3,1,5))
## check dimension
B
dim(B)
```

```
     [,1] [,2] [,3]
[1,]    2    5    2
[2,]    6    1    4
[1] 2 3
     [,1] [,2]
[1,]    3   -3
[2,]   -1    1
[3,]    1    5
[1] 3 2
```

- Let's write a function `multiply` to check compatibility and run it:

  1. `function` arguments are matrices `A` and `B`.
  2. test if `ncol(A)` is equal to `nrow(B)`
  3. multiply if `TRUE`, otherwise print message
  4. test the function on the globally defined `A` and `B` matrices

```
## define function with args A and B
multiply <- function(A,B) {
  ## check if A and B are compatible for multiplication
  ## if so, return their product
  ## otherwise, let the user know (why)
  if(ncol(A) == nrow(B)) {
    return (A %*% B)
  } else {
    paste("Matrices not compatible!")
  }
}
## test your function with the previously defined matrices
A <- rbind(c(2,5,2),  # rbind is filled by row
           c(6,1,4))
A
## check dimension
dim(A)
B <- cbind(c(3,-1,1),  # cbind is filled by column
           c(-3,1,5))
## check dimension
B
dim(B)
multiply(A,B)
```

```
     [,1] [,2] [,3]
[1,]    2    5    2
[2,]    6    1    4
[1] 2 3
     [,1] [,2]
[1,]    3   -3
[2,]   -1    1
[3,]    1    5
[1] 3 2
```

```
          [,1] [,2]
     [1,]    3    9
     [2,]   21    3
```

- Solution for `multiply`:

```
multiply_ <- function(A,B) {
  if (ncol(A) == nrow(B)) {
    A %*%B
  } else {
    paste("Not compatible:", ncol(A), "!=", nrow(B))
  }
}
A <- rbind(c(2,5,2),  # rbind is filled by row
           c(6,1,4))
A
## check dimension
dim(A)
B <- cbind(c(3,-1,1),  # cbind is filled by column
           c(-3,1,5))
## check dimension
B
dim(B)
multiply_(A,B)

     [,1] [,2] [,3]
[1,]    2    5    2
[2,]    6    1    4
[1] 2 3
     [,1] [,2]
[1,]    3   -3
[2,]   -1    1
[3,]    1    5
[1] 3 2
     [,1] [,2]
[1,]    3    9
[2,]   21    3
```

- Matrix multiplication is generally not commutative. Check that by running `multiply` on `B` and `A` in reverse order from before:

```
## multiply B and A
multiply(B,A)
multiply(A,B)

     [,1] [,2] [,3]
[1,]  -12   12   -6
[2,]    4   -4    2
[3,]   32   10   22
     [,1] [,2]
[1,]    3    9
[2,]   21    3
```

- Finally, test the function on two incompatible matrices `C` and `D`:

```
C <- matrix(1:4,2); C
D <- matrix(1:9,3); D
## multiply C and D
multiply_(C,D)

     [,1] [,2]
[1,]    1    3
[2,]    2    4
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
[1] "Not compatible: 2 != 3"
```

Example: multiple `function` arguments

- Write a function `mult1` that accepts as arguments:
  1. a list `x`
  2. a matrix `mat`
  3. two strings `str1` and `str2`
- The function will search through `x`, look for matrices that can be multiplied with `mat`, and store the return the result in a new list.

- If no (compatible) matrices are in the supplied list `x`, the user should be informed of these facts.
- Pseudocode for the function body (objects marked with `$`):

```
$flag matrices in the list $x
If $x contains no matrices
   return $str1
Otherwise:
   Make index from $flag
   Initialize matrix $counter and $result list
   Loop over matrices
     Store matrix in $temp
     If matrix is compatible with mat
        Increase matrix counter by 1
        Multiply $temp and $mat, store in $result
   If $counter is 0 (no compatible matrices)
     return $str2
   Otherwise:
     return $result of matrix multiplication
```

- Let's code the function `mult1` with arguments `x`, `mat`, `str1`, `str2`:

```
mult1 <- function(x, mat, str1, str2) {
  ## $flag matrices - use 'sapply', FUN=is.matrix
  flag <- sapply(X=x, FUN=is.matrix)
  ## check if $x has 'any' matrices, otherwise 'return' $str1
  if (!any(flag))  { # TRUE if there a no matrices in x
    return(str1)
  }
  ## $x contains matrices! make index vector $idx from $flag
  idx <- which(flag) # TRUE for matrix elements of x
  ## initialize matrix $counter to 0 and an empty $result list
  counter <- 0
  result <- list()
  ## loop over matrices (use $idx as loopindex)
  for (i in idx) { # sets i to index of x that contains a matrix
    ## store $x in $temp
    x[[i]] -> temp
    ## check if dim of $x and $mat are compatible if TRUE, multiply
```

```
      ## them, store in result (index counter) and increase counter
      if (ncol(temp) == nrow(mat)) {
        counter + 1 -> counter
        temp %*% mat -> result[[counter]]
      }
    } ## end of loop over matrix elements of list x
    ## check if $counter is still 0 then 'return' $str2
    if (counter == 0) { ## if TRUE then none of the matrices compatible
      return(str2)
    } else {      ## otherwise 'return' $result
      return (result)
    }
  }
```

- Solution:

```
mult1 <- function(x,mat,str1,str2) {
  flag <- sapply(x, FUN=is.matrix)
  if(!any(flag)) return (str1)
  idx <- which(flag)
  counter <- 0
  result <- list()
  for (i in idx) {
    x[[i]] -> temp
    if (ncol(temp) == nrow(mat)) {
      counter <- counter + 1
      temp %*% mat -> result[[counter]]
    }
  }
  if (counter == 0) {
    return (str2)
  } else {
    return (result)
  }
}
```

- Test suite with three `list` objects `foo`, `bar` and `baz`

```
foo <- list(matrix(1:4,2,2),
            "not a matrix",
            "definitely not a matrix",
            matrix(1:8,2,4),
            matrix(1:8,4,2))
bar <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??")
baz <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??",
            matrix(1:8,2,4))
```

- Test `mult1` with `foo` and set `mat` to the 2 x 2 identity matrix - so that post-multiplying any matrix with `mat` will simply return the original matrix, as well as appropriate messages `str1`, `str2`:

```
foo <- list(matrix(1:4,2,2),
            "not a matrix",
            "definitely not a matrix",
            matrix(1:8,2,4),
            matrix(1:8,4,2))
bar <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??")
baz <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??",
            matrix(1:8,2,4))
mult1 <- function(x, mat, str1, str2) {
  ## $flag matrices - use 'sapply', FUN=is.matrix
  flag <- sapply(X=x, FUN=is.matrix)
  ## check if $x has 'any' matrices, otherwise 'return' $str1
  if (!any(flag))  { # TRUE if there a no matrices in x
    return(str1)
  }
  ## $x contains matrices! make index vector $idx from $flag
  idx <- which(flag) # TRUE for matrix elements of x
  ## initialize matrix $counter to 0 and an empty $result list
  counter <- 0
  result <- list()
  ## loop over matrices (use $idx as loopindex)
  for (i in idx) { # sets i to index of x that contains a matrix
    ## store $x in $temp
    x[[i]] -> temp
    ## check if dim of $x and $mat are compatible if TRUE, multiply
    ## them, store in result (index counter) and increase counter
    if (ncol(temp) == nrow(mat)) {
      counter + 1 -> counter
      temp %*% mat -> result[[counter]]
    }
  } ## end of loop over matrix elements of list x
  ## check if $counter is still 0 then 'return' $str2
  if (counter == 0) { ## if TRUE then none of the matrices compatible
    return(str2)
  } else {      ## otherwise 'return' $result
    return (result)
  }
}
mult1(x = foo,
      mat = diag(2),  # 2 x 2 identity matrix
      str1 = "no matrices in x",
      str2 = "no compatible matrices in x")

[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[2]]
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

- Test `mult1` with `bar`, which has no matrices at all, and the same arguments otherwise:

```
foo <- list(matrix(1:4,2,2),
            "not a matrix",
            "definitely not a matrix",
            matrix(1:8,2,4),
            matrix(1:8,4,2))
bar <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??")
baz <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??",
            matrix(1:8,2,4))
mult1 <- function(x, mat, str1, str2) {
  ## $flag matrices - use 'sapply', FUN=is.matrix
  flag <- sapply(X=x, FUN=is.matrix)
  ## check if $x has 'any' matrices, otherwise 'return' $str1
  if (!any(flag))  { # TRUE if there a no matrices in x
    return(str1)
  }
  ## $x contains matrices! make index vector $idx from $flag
  idx <- which(flag) # TRUE for matrix elements of x
  ## initialize matrix $counter to 0 and an empty $result list
  counter <- 0
  result <- list()
  ## loop over matrices (use $idx as loopindex)
  for (i in idx) { # sets i to index of x that contains a matrix
    ## store $x in $temp
    x[[i]] -> temp
    ## check if dim of $x and $mat are compatible if TRUE, multiply
    ## them, store in result (index counter) and increase counter
    if (ncol(temp) == nrow(mat)) {
      counter + 1 -> counter
      temp %*% mat -> result[[counter]]
    }
  } ## end of loop over matrix elements of list x
  ## check if $counter is still 0 then 'return' $str2
  if (counter == 0) { ## if TRUE then none of the matrices compatible
    return(str2)
  } else {      ## otherwise 'return' $result
    return (result)
  }
}
mult1(x = bar,
      mat = diag(2),
      str1 = "no matrices in x at all!!!!!",
      str2 = "no compatible matrices in x")

[1] "no matrices in x at all!!!!!"
```

- Finally, test `mult1` with `baz`, which has one matrix but no compatibility for multiplication with `mat`:

```
foo <- list(matrix(1:4,2,2),
            "not a matrix",
            "definitely not a matrix",
            matrix(1:8,2,4),
            matrix(1:8,4,2))
bar <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??")
baz <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??",
            matrix(1:8,2,4))
mult1 <- function(x, mat, str1, str2) {
  ## $flag matrices - use 'sapply', FUN=is.matrix
  flag <- sapply(X=x, FUN=is.matrix)
  ## check if $x has 'any' matrices, otherwise 'return' $str1
  if (!any(flag))  { # TRUE if there a no matrices in x
    return(str1)
  }
  ## $x contains matrices! make index vector $idx from $flag
  idx <- which(flag) # TRUE for matrix elements of x
  ## initialize matrix $counter to 0 and an empty $result list
  counter <- 0
  result <- list()
  ## loop over matrices (use $idx as loopindex)
  for (i in idx) { # sets i to index of x that contains a matrix
    ## store $x in $temp
    x[[i]] -> temp
    ## check if dim of $x and $mat are compatible if TRUE, multiply
    ## them, store in result (index counter) and increase counter
    if (ncol(temp) == nrow(mat)) {
      counter + 1 -> counter
      temp %*% mat -> result[[counter]]
    }
  } ## end of loop over matrix elements of list x
  ## check if $counter is still 0 then 'return' $str2
  if (counter == 0) { ## if TRUE then none of the matrices compatible
    return(str2)
  } else {      ## otherwise 'return' $result
    return (result)
  }
}
mult1(x = baz,
      mat = diag(2),
      str1 = "no matrices in x",
      str2 = "no compatible matrices in x at all!!")
```

```
[1] "no compatible matrices in x at all!!"
```

- Notice that the string arguments `str1` and `str2` are used only when the argument `x` does not contain a matrix with the appropriate dimensions.
- R evaluates the arguments "lazily": argument values are sought only when they are required during execution. For `x=foo` you could lazily ignore the string arguments.
- Run `mult1` again only for `x` and `mat`:

```
foo <- list(matrix(1:4,2,2),
            "not a matrix",
            "definitely not a matrix",
            matrix(1:8,2,4),
            matrix(1:8,4,2))
bar <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??")
baz <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??",
            matrix(1:8,2,4))
mult1 <- function(x, mat, str1, str2) {
  ## $flag matrices - use 'sapply', FUN=is.matrix
  flag <- sapply(X=x, FUN=is.matrix)
  ## check if $x has 'any' matrices, otherwise 'return' $str1
  if (!any(flag))  { # TRUE if there a no matrices in x
    return(str1)
  }
  ## $x contains matrices! make index vector $idx from $flag
  idx <- which(flag) # TRUE for matrix elements of x
  ## initialize matrix $counter to 0 and an empty $result list
  counter <- 0
  result <- list()
  ## loop over matrices (use $idx as loopindex)
  for (i in idx) { # sets i to index of x that contains a matrix
    ## store $x in $temp
    x[[i]] -> temp
    ## check if dim of $x and $mat are compatible if TRUE, multiply
    ## them, store in result (index counter) and increase counter
    if (ncol(temp) == nrow(mat)) {
      counter + 1 -> counter
      temp %*% mat -> result[[counter]]
    }
  } ## end of loop over matrix elements of list x
  ## check if $counter is still 0 then 'return' $str2
  if (counter == 0) { ## if TRUE then none of the matrices compatible
    return(str2)
  } else {      ## otherwise 'return' $result
    return (result)
  }
}
mult1(foo,diag(2))
```

```
[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4

[[2]]
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
```

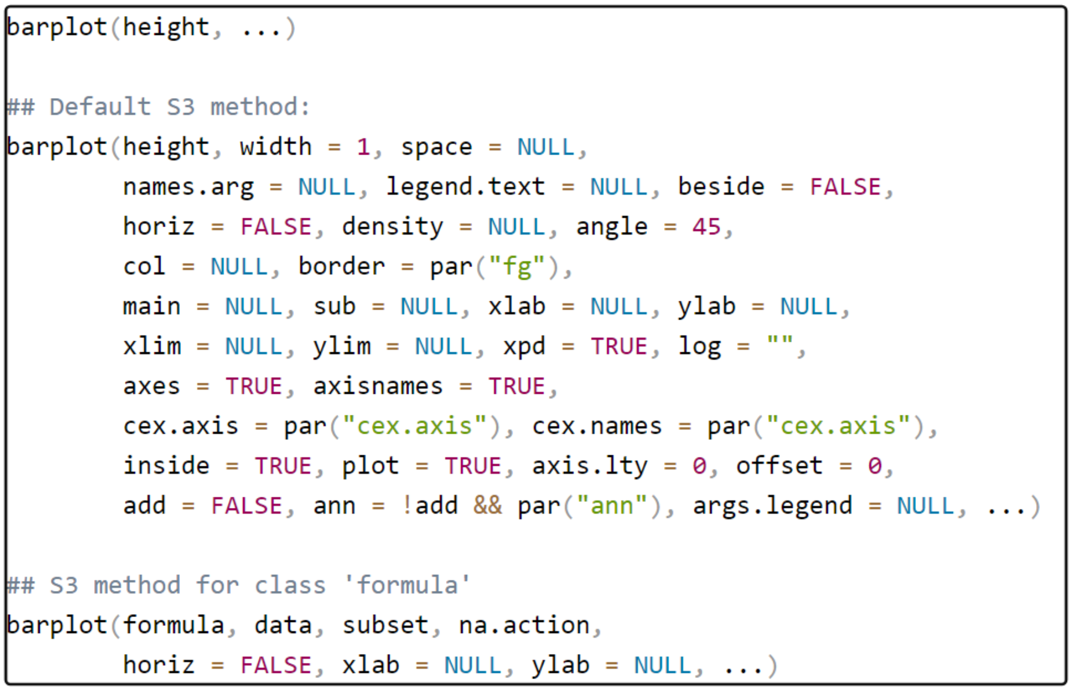
- However, for `x=bar` this will not work - an argument is missing:

```
foo <- list(matrix(1:4,2,2),
            "not a matrix",
            "definitely not a matrix",
            matrix(1:8,2,4),
            matrix(1:8,4,2))
bar <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??")
baz <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??",
            matrix(1:8,2,4))
mult1 <- function(x, mat, str1, str2) {
  ## $flag matrices - use 'sapply', FUN=is.matrix
  flag <- sapply(X=x, FUN=is.matrix)
  ## check if $x has 'any' matrices, otherwise 'return' $str1
  if (!any(flag))  { # TRUE if there a no matrices in x
    return(str1)
  }
  ## $x contains matrices! make index vector $idx from $flag
  idx <- which(flag) # TRUE for matrix elements of x
  ## initialize matrix $counter to 0 and an empty $result list
  counter <- 0
  result <- list()
  ## loop over matrices (use $idx as loopindex)
  for (i in idx) { # sets i to index of x that contains a matrix
    ## store $x in $temp
    x[[i]] -> temp
    ## check if dim of $x and $mat are compatible if TRUE, multiply
    ## them, store in result (index counter) and increase counter
    if (ncol(temp) == nrow(mat)) {
      counter + 1 -> counter
      temp %*% mat -> result[[counter]]
    }
  } ## end of loop over matrix elements of list x
  ## check if $counter is still 0 then 'return' $str2
  if (counter == 0) { ## if TRUE then none of the matrices compatible
    return(str2)
  } else {      ## otherwise 'return' $result
    return (result)
  }
}
mult1(x=bar,mat=diag(2))

Error in mult1(x = bar, mat = diag(2)) :
  argument "str1" is missing, with no default
```

Setting default arguments

- In the previous example, a default argument would have been useful to cover one of the outcomes.
- Default arguments are also useful when arguments have a large number of natural values that are routinely used.
- Many R functions have such default values, see e.g. `barplot`:



```
barplot(height, ...)

## Default S3 method:
barplot(height, width = 1, space = NULL,
        names.arg = NULL, legend.text = NULL, beside = FALSE,
        horiz = FALSE, density = NULL, angle = 45,
        col = NULL, border = par("fg"),
        main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
        xlim = NULL, ylim = NULL, xpd = TRUE, log = "",
        axes = TRUE, axisnames = TRUE,
        cex.axis = par("cex.axis"), cex.names = par("cex.axis"),
        inside = TRUE, plot = TRUE, axis.lty = 0, offset = 0,
        add = FALSE, ann = !add && par("ann"), args.legend = NULL, ...)

## S3 method for class 'formula'
barplot(formula, data, subset, na.action,
        horiz = FALSE, xlab = NULL, ylab = NULL, ...)
```

- `barplot` has different methods depending on the class of data fed into it. Can you see how many mandatory arguments each method has?
- Create another version of `mult1` and name it `mult2`, which includes default values for `str1` and `str2`.
- Below is the code for `mult1` with the new name - add the default values yourself:

```
mult2 <- function(x,mat,str1,str2) {
  flag <- sapply(x, FUN=is.matrix)
  if(!any(flag)) return (str1)
  idx <- which(flag)
  counter <- 0
  result <- list()
  for (i in idx) {
    x[[i]] -> temp
    if (ncol(temp) == nrow(mat)) {
      counter <- counter + 1
      temp %*% mat -> result[[counter]]
    }
  }
  if (counter == 0) {
    return (str2)
  } else {
    return (result)
  }
}
```

- Now re-run the test suite for the three lists `foo`, `bar`, and `baz`, with `mat` as the 2 x 2 identity matrix as before:

```
foo <- list(matrix(1:4,2,2),
            "not a matrix",
            "definitely not a matrix",
            matrix(1:8,2,4),
            matrix(1:8,4,2))
bar <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??")
baz <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??",
            matrix(1:8,2,4))
mult2 <- function(x,mat,str1,str2) {
  flag <- sapply(x, FUN=is.matrix)
  if(!any(flag)) return (str1)
  idx <- which(flag)
  counter <- 0
  result <- list()
  for (i in idx) {
    x[[i]] -> temp
    if (ncol(temp) == nrow(mat)) {
      counter <- counter + 1
      temp %*% mat -> result[[counter]]
    }
  }
  if (counter == 0) {
    return (str2)
  } else {
    return (result)
  }
}
## test foo
## test bar
## test baz
```

- Solution:

```
mult2_ <- function(x, mat,
                   str1="No matrices in list",
                   str2="No compatible matrices in list") {
  flag <- sapply(x, FUN=is.matrix)
  if(!any(flag)) return (str1)
  idx <- which(flag)
  counter <- 0
  result <- list()
  for (i in idx) {
    x[[i]] -> temp
    if (ncol(temp) == nrow(mat)) {
      counter <- counter + 1
      temp %*% mat -> result[[counter]]
    }
  }
  if (counter == 0) {
    return (str2)
  } else {
    return (result)
  }
}
```

- Solution (test suite):

```
foo <- list(matrix(1:4,2,2),
            "not a matrix",
            "definitely not a matrix",
            matrix(1:8,2,4),
            matrix(1:8,4,2))
bar <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??")
baz <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??",
            matrix(1:8,2,4))
mult2_ <- function(x, mat,
                   str1="No matrices in list",
                   str2="No compatible matrices in list") {
  flag <- sapply(x, FUN=is.matrix)
  if(!any(flag)) return (str1)
  idx <- which(flag)
  counter <- 0
  result <- list()
  for (i in idx) {
    x[[i]] -> temp
    if (ncol(temp) == nrow(mat)) {
      counter <- counter + 1
      temp %*% mat -> result[[counter]]
    }
  }
  if (counter == 0) {
    return (str2)
  } else {
    return (result)
  }
}
mult2_(x = foo,
       mat = diag(2))
mult2_(x = bar,
       mat = diag(2),str1="I did not mean that!")
mult2_(x = baz,
       mat = diag(2))

[[1]]
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

```
[[2]]
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8
[1] "I did not mean that!"
[1] "No compatible matrices in list"
```

- If you do not want to use the default, you can override it. Call `mult2` again for `baz` and change the argument for `str2` alone to: "Matrices in baz do not have 2 columns."

```
foo <- list(matrix(1:4,2,2),
            "not a matrix",
            "definitely not a matrix",
            matrix(1:8,2,4),
            matrix(1:8,4,2))
bar <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??")
baz <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??",
            matrix(1:8,2,4))
mult2 <- function(x,mat,str1,str2) {
  flag <- sapply(x, FUN=is.matrix)
  if(!any(flag)) return (str1)
  idx <- which(flag)
  counter <- 0
  result <- list()
  for (i in idx) {
    x[[i]] -> temp
    if (ncol(temp) == nrow(mat)) {
      counter <- counter + 1
      temp %*% mat -> result[[counter]]
    }
  }
  if (counter == 0) {
    return (str2)
  } else {
    return (result)
  }
}
## test baz but specify argument str2
```

- Solution:

```
foo <- list(matrix(1:4,2,2),
            "not a matrix",
            "definitely not a matrix",
            matrix(1:8,2,4),
            matrix(1:8,4,2))
bar <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??")
baz <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??",
            matrix(1:8,2,4))
mult2_ <- function(x, mat,
                   str1="No matrices in list",
                   str2="No compatible matrices in list") {
  flag <- sapply(x, FUN=is.matrix)
  if(!any(flag)) return (str1)
  idx <- which(flag)
  counter <- 0
  result <- list()
  for (i in idx) {
    x[[i]] -> temp
    if (ncol(temp) == nrow(mat)) {
      counter <- counter + 1
      temp %*% mat -> result[[counter]]
    }
  }
  if (counter == 0) {
    return (str2)
  } else {
    return (result)
  }
}
## test baz but specify argument str2
mult2_(baz,diag(2),
       str2="Matrices in baz do not have 2 columns.")

[1] "Matrices in baz do not have 2 columns."
```

Checking for missing arguments

- The `missing` function checks the arguments of a function to see if all required arguments have been supplied.
- The function takes an argument tag and returns `TRUE` if the specified argument is not found.
- Example using the `hello(name)` function: the function `hello` throws an error when called without argument.

  ```
  hello <- function(name) {
    return(paste("Hello",name))
  }
  hello()

  Error in paste("Hello", name) :
    argument "name" is missing, with no default
  ```

- But this function with a check and call to `missing` will not break; add the missing argument check and test the function `hello_1`:

  ```
  hello_1 <- function(name) {
    return(paste("Hello",name))
  }
  ## test function hello_1 without arguments
  ```

- Solution:

```
hello_ <- function(name) {
  if (missing(name)) {
    return("'name' was missing, so this is the message")
  } else {
    return(paste("Hello",name))
  }
}
hello_()
hello_("Marcus")

[1] "'name' was missing, so this is the message"
[1] "Hello Marcus"
```

- This takes care of the error encountered earlier in the call to `mult1`, when `str1` was required but not found (because no default had been set).
- In the modification `mult3` of the algorithm `mult1`, add an argument check with missing both for `str1` and `str2`.

```
mult3 <- function(x,mat,str1,str2) {
  flag <- sapply(x, FUN=is.matrix)
  if(!any(flag)) return (str1)
  idx <- which(flag)
  counter <- 0
  result <- list()
  for (i in idx) {
    x[[i]] -> temp
    if (ncol(temp) == nrow(mat)) {
      counter <- counter + 1
      temp %*% mat -> result[[counter]]
    }
  }
  if (counter == 0) {
    return (str2)
  } else {
    return (result)
  }
}
```

- Test `mult3` with `bar` (which returns `str1`) and with `baz` (which returns `str2`:

  ```
  foo <- list(matrix(1:4,2,2),
              "not a matrix",
              "definitely not a matrix",
              matrix(1:8,2,4),
              matrix(1:8,4,2))
  bar <- list(1:4,
              "not a matrix",
              c(F,T,T,T),
              "??")
  baz <- list(1:4,
              "not a matrix",
              c(F,T,T,T),
              "??",
              matrix(1:8,2,4))
  mult3 <- function(x,mat,str1,str2) {
    flag <- sapply(x, FUN=is.matrix)
    if(!any(flag)) return (str1)
    idx <- which(flag)
    counter <- 0
    result <- list()
    for (i in idx) {
      x[[i]] -> temp
      if (ncol(temp) == nrow(mat)) {
        counter <- counter + 1
        temp %*% mat -> result[[counter]]
      }
    }
    if (counter == 0) {
      return (str2)
    } else {
      return (result)
    }
  }
  ## test with bar
  ## test with baz
  ```

- The `missing` function is also useful when it it difficult to choose a default value for a certain argument, yet the function needs to handle cases when that argument is not provided.

- In the case of this algorithm, it makes more sense to define defaults for `str1` and `str2` and avoid the overhead of `missing`.
- Solution (`mult3`):

```
mult3_ <- function(x, mat, str1, str2) {
  flag <- sapply(x,is.matrix)
  if(!any(flag)) {
    if(missing(str1)) {
      return ("'str1' was missing, so this is the message.")
    } else {
      return(str1)
    }
  }
  idx <- which(flag)
  counter <- 0
  result <- list()
  for (i in idx) {
    x[[i]] -> temp
    if (ncol(temp) == nrow(mat)) {
      counter <- counter + 1
      temp %*% mat -> result[[counter]]
    }
  }
  if(counter==0) {
    if (missing(str2)) {
      return("'str2' was missing, so this is the message")
    } else {
      return(str2)
    }
  } else {
    return (result)
  }
}
```

- Solution (test of `mult3`):

```
foo <- list(matrix(1:4,2,2),
            "not a matrix",
            "definitely not a matrix",
            matrix(1:8,2,4),
            matrix(1:8,4,2))
bar <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??")
baz <- list(1:4,
            "not a matrix",
            c(F,T,T,T),
            "??",
            matrix(1:8,2,4))
mult3_ <- function(x, mat, str1, str2) {
  flag <- sapply(x,is.matrix)
  if(!any(flag)) {
    if(missing(str1)) {
      return ("'str1' was missing, so this is the message.")
    } else {
      return(str1)
    }
  }
  idx <- which(flag)
  counter <- 0
  result <- list()
  for (i in idx) {
    x[[i]] -> temp
    if (ncol(temp) == nrow(mat)) {
      counter <- counter + 1
      temp %*% mat -> result[[counter]]
    }
  }
  if(counter==0) {
    if (missing(str2)) {
      return("'str2' was missing, so this is the message")
    } else {
      return(str2)
    }
  } else {
    return (result)
  }
}
mult3_(bar,diag(2))  # needs str1 and str2
mult3_(baz,diag(2))  # needs str2 and str2

[1] "'str1' was missing, so this is the message."
[1] "'str2' was missing, so this is the message"
```

- How does `missing` look like?

```
missing

function (x)  .Primitive("missing")
```

Dealing with ellipses

- The *ellipsis* or *dot-dot-dot* notation allows you to pass extra arguments without having to first define them in the argument list.
- In a `function` definition, it is placed in the last position representing a variable number of arguments.
- In the example, we use the ellipsis to write a function that can plot the specified Fibonacci numbers:
  1. create the Fibonacci sequence `fibseq` to `threshold`
  2. if `plotit` is `TRUE`, plot the obtained sequence on the y-axis against its index for the x-axis
  3. Pass the ellipsis right into `plot`

```
myfibplot <- function(threshold, plotit=TRUE,...) {
  fibseq <- c(1,1)  # initialize
  counter <- 2
  repeat {
    fibseq <- c(fibseq,fibseq[counter-1]+fibseq[counter])
    counter <- counter +1
    if (fibseq[counter] > threshold) break
  }
  if (plotit) {
    plot(x=1:length(fibseq),y=fibseq,...)  # ellipsis ...
  } else {
    return (fibseq)
  }
}
```

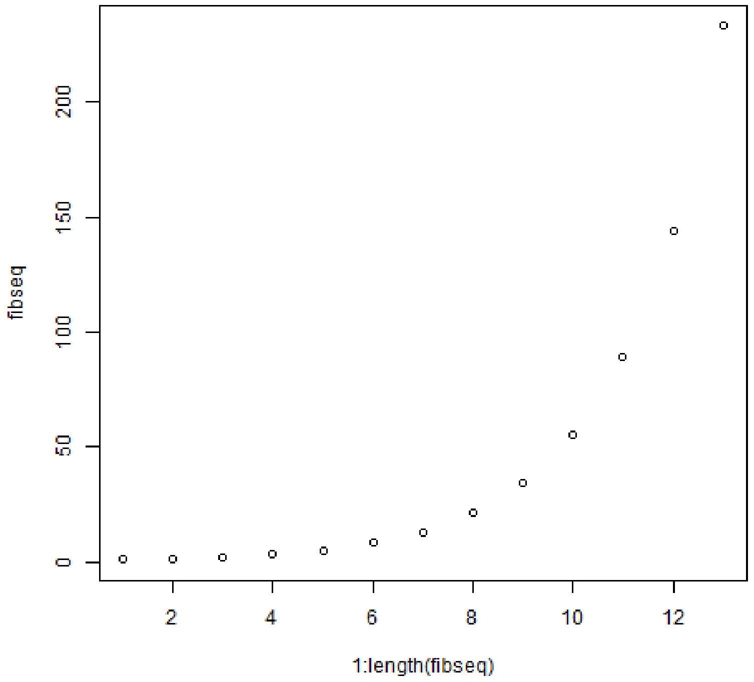
- Suppress the plot but print the sequence up to `threshold=150`:

```
myfibplot <- function(threshold, plotit=TRUE,...) {
  fibseq <- c(1,1)  # initialize
  counter <- 2
  repeat {
    fibseq <- c(fibseq,fibseq[counter-1]+fibseq[counter])
    counter <- counter +1
    if (fibseq[counter] > threshold) break
  }
  if (plotit) {
    plot(x=1:length(fibseq),y=fibseq,...)  # ellipsis ...
  } else {
    return (fibseq)
  }
}
myfibplot(threshold=150,plotit=FALSE)

[1]   1   1   2   3   5   8  13  21  34  55  89 144 233
```

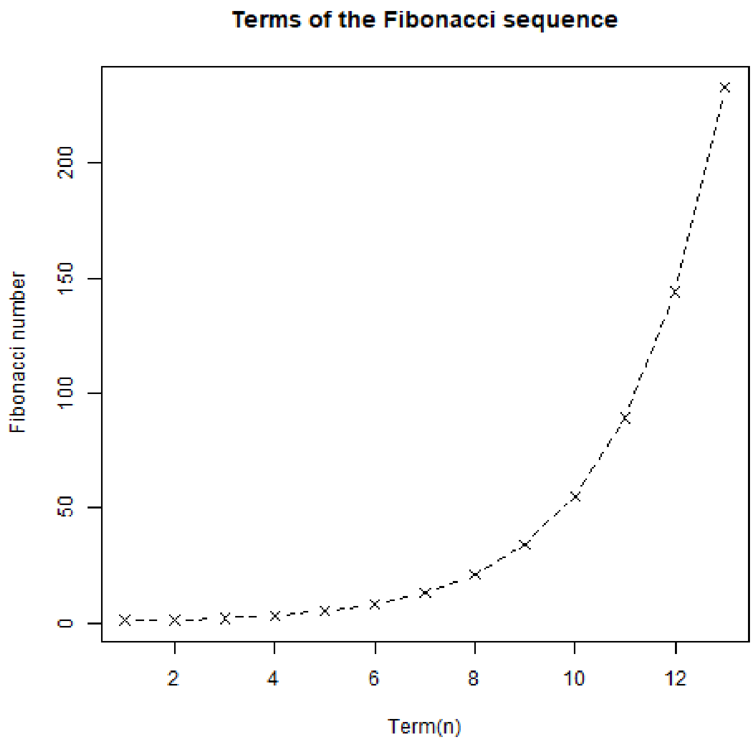
- Plot Fibonacci numbers up to the `threshold` 150 with `myfibplot`:

```
myfibplot <- function(threshold, plotit=TRUE,...) {
  fibseq <- c(1,1)  # initialize
  counter <- 2
  repeat {
    fibseq <- c(fibseq,fibseq[counter-1]+fibseq[counter])
    counter <- counter +1
    if (fibseq[counter] > threshold) break
  }
  if (plotit) {
    plot(x=1:length(fibseq),y=fibseq,...)  # ellipsis ...
  } else {
    return (fibseq)
  }
}
myfibplot(150)
```



- In this plot, the ellipsis is not used. In the next one, we'll use it. Add the following arguments to the function call:
  1. Give it the title "Terms of the Fibonacci sequence"
  2. Change the point character (`pch`) to 4
  3. Change the line type (`lty`) to 2
  4. Change the x-axis label (`xlab`) to "Term (n)"
  5. Change the y-axis label (`ylab`) to "Fibonacci number"
  6. Change the plot `type` to "both points and lines" (`"b"`)

```
myfibplot <- function(threshold, plotit=TRUE,...) {
  fibseq <- c(1,1)  # initialize
  counter <- 2
  repeat {
    fibseq <- c(fibseq,fibseq[counter-1]+fibseq[counter])
    counter <- counter +1
    if (fibseq[counter] > threshold) break
  }
  if (plotit) {
    plot(x=1:length(fibseq),y=fibseq,...)  # ellipsis ...
  } else {
    return (fibseq)
  }
}
myfibplot(150,
          main="Terms of the Fibonacci sequence",
          pch=4,
          lty=2,
          xlab="Term(n)",
          ylab="Fibonacci number",
          type="b")
```



- Solution `myfibplot` function:

```
myfibplot_ <- function(threshold, plotit=TRUE,...) {
  fibseq <- c(1,1)  # initialize
  counter <- 2
  repeat {
    fibseq <- c(fibseq,fibseq[counter-1]+fibseq[counter])
    counter <- counter +1
    if (fibseq[counter] > threshold) break
  }
  if (plotit) {
    plot(x=1:length(fibseq),
         y=fibseq, ...)
  } else {
    return (fibseq)
  }
}
```

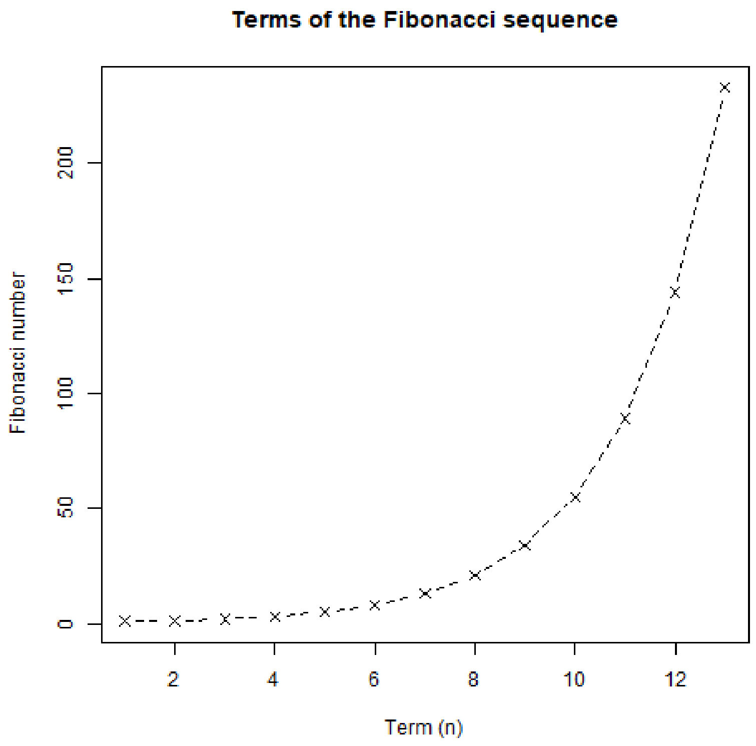
- Solution to suppress the `plot` but print numbers with `threshold` 150:

```
myfibplot_ <- function(threshold, plotit=TRUE,...) {
  fibseq <- c(1,1)  # initialize
  counter <- 2
  repeat {
    fibseq <- c(fibseq,fibseq[counter-1]+fibseq[counter])
    counter <- counter +1
    if (fibseq[counter] > threshold) break
  }
  if (plotit) {
    plot(x=1:length(fibseq),
         y=fibseq, ...)
  } else {
    return (fibseq)
  }
}
myfibplot_(threshold=150, plotit=FALSE)
```

```
[1]   1   1   2   3   5   8  13  21  34  55  89 144 233
```

- Solution for adding `plot` arguments:

```
myfibplot_ <- function(threshold, plotit=TRUE,...) {
  fibseq <- c(1,1)  # initialize
  counter <- 2
  repeat {
    fibseq <- c(fibseq,fibseq[counter-1]+fibseq[counter])
    counter <- counter +1
    if (fibseq[counter] > threshold) break
  }
  if (plotit) {
    plot(x=1:length(fibseq),
         y=fibseq, ...)
  } else {
    return (fibseq)
  }
}
myfibplot_(threshold = 150,
           main="Terms of the Fibonacci sequence",
        pch=4,
        lty=2,
        xlab="Term (n)",
        ylab="Fibonacci number",
        type="b")
```



- The ellipsis `...` can represent any number of mysterious arguments - to indicate proper usage, document your functions well.
- The dummy function `unpackme(...)` takes an ellipsis converts it into a list.

```
unpackme <- function(...) {
  x <- list(...)
  cat("Here is ... in its entirety as a list:\n")
  print(x)
  cat("The names of ... are:", names(x),"\n")
  cat("The classes of ... are:", unname(unlist(sapply(x,class))),"\n")
}
```

- The short version:

```
unpackme <- function(...) {
  x <- list(...)
  print(x)  # print ellipsis elements
  print(names(x))  # print ellipsis element names
  print(unname(unlist(sapply(x,class)))) # print ellipsis element types
}
```

- Here's a sample run:

  1. four arguments, tagged `aa`, `bb`, `cc`, and `dd` are contents of ...
  2. `unpackme` identifies them using `list`

```
unpackme <- function(...) {
  x <- list(...)
  cat("Here is ... in its entirety as a list:\n")
  print(x)
  cat("The names of ... are:", names(x),"\n")
  cat("The classes of ... are:", unname(unlist(sapply(x,class))),"\n")
}
unpackme(aa = matrix(1:4,2,2),
         bb = TRUE,
         cc = c("two","strings"),
         dd = factor(c(1,1,2,1)))

Here is ... in its entirety as a list:
$aa
     [,1] [,2]
[1,]    1    3
[2,]    2    4

$bb
[1] TRUE

$cc
[1] "two"     "strings"
```

```
$dd
[1] 1 1 2 1
Levels: 1 2

The names of ... are: aa bb cc dd
The classes of ... are: matrix array logical character factor
```

TODO Exercises

These bonus exercises carry 20 points each for successful completion. These exercises come from Davies, ch. 11. pp. 230-231.

Annual compound interest function

Accruing annual compound interest is a common financial benefit for investors. Given a principal investment amoung P, an interest rate per annum i expressed as a percentage, and a frequency of interest paid per year t, the final amount F after y years is given in this formula:

$$F = P\left(1 + \frac{i}{100t}\right)^{ty}$$

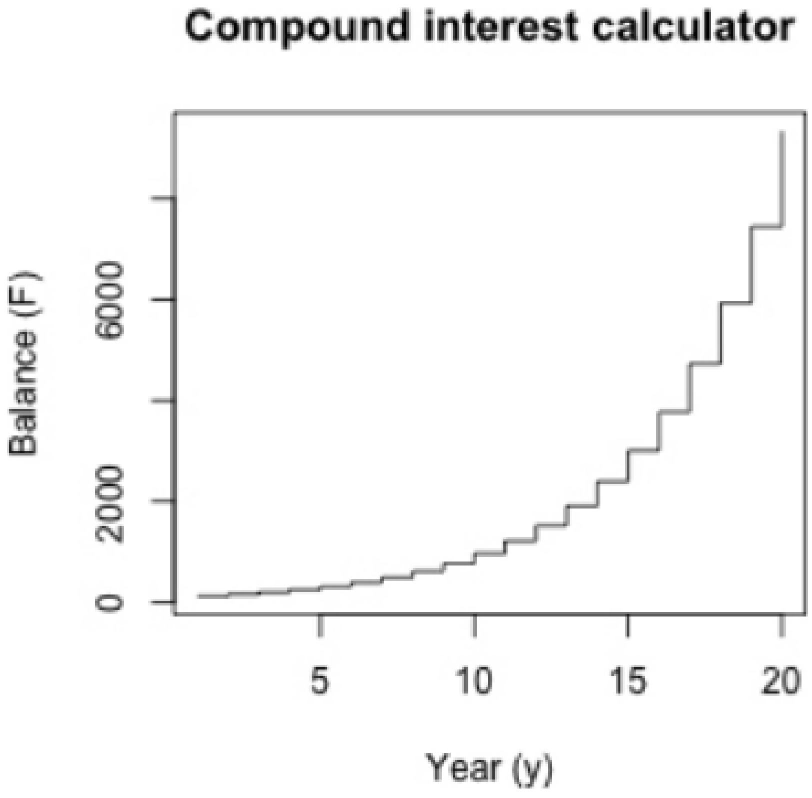
Write a function `F` to compute F as per these instructions:

- Arguments must be present for P, i, t and y. The argument for t should have a default value of 12.
- Another argument giving a logical value that determines whether to `plot` the amount F at each integer time should be included. For example, if `plotit=TRUE` (the default) and `y` is 5 years, the plot should show the amount F at y = 1,2,3,4,5.
- If this function is plotted, the plot should always be a step-plot, so `plot` should always be called with `type="s"`.
- If `plotit=FALSE`, the final amount F should be returned as a `numeric` vector corresponding to the same integer times, as shown earlier.

- An ellipsis should also be included to control other details of plotting, if it takes place.

Using your function `F`, do the following:

1. Work out the final amount after a 10-year investment of a principal of \$5000, at an interest rate of 4.4 percent per annum compounded monthly.
2. Re-create the following step-plot, which shows the result of \$100 invested at 22.9 percent per annum, compounded monthly, for 20 years:

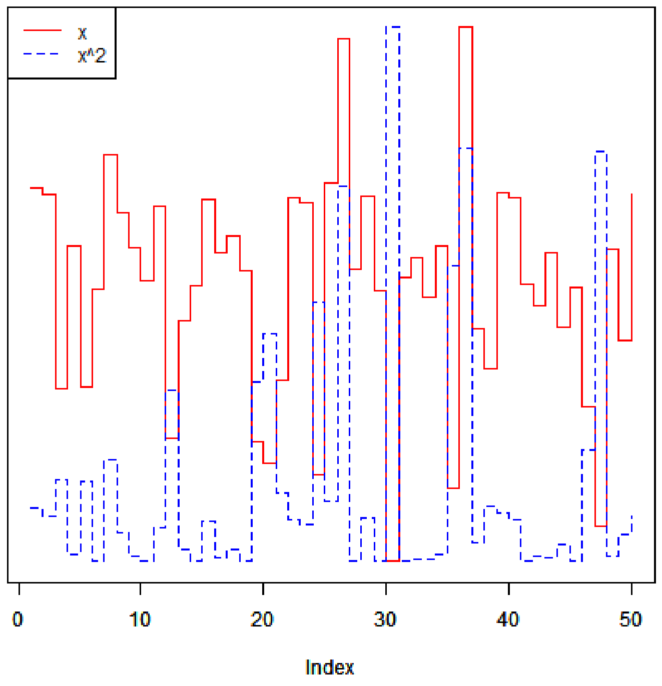


3. Perform another calculation based on the same parameters as in (2), but this time, assume the interest is compounded annually. Return and store the results as a numeric vector. Then, use `lines` to add a second step-line, corresponding to this annually accrued amount, to the plot created previously. Use a different color or line type and make use of the `legend` function so the two lines can be differentiated.

   Tip - this code prints two step-wise plots in the same graph:

- the first `plot` plots 50 random numbers `x`
- the second `plot` plots the squared values of `x`

```
par(mfrow=c(1,1))
x <- rnorm(50)
plot(x,type='s', col="red", yaxt="n", ylab="")
par(new=TRUE)
plot(x^2,type='s', col="blue", lty=2, ylab="", yaxt="n")
legend("topleft",
       legend=c("x","x^2"), lty=c(1,2),col=c("red","blue"))
```



Real roots of a quadratic equation

(Note: to show the formulae in mathematical form, simply add this to your Org-mode header (and run it with C-c C-c): #+STATUS: entitiespretty

A quadratic equation in the variable x is often expressed in the following form: k_{1}x^{2} + k_{2}x + k_{3} = 0. Here, k_{1}, k_{2} and k_{3} are constants. Given values for these constants, you can attempt to find up to two real roots - values of x that satisfy the equation.

Write a function that takes k_{1}, k_{2} and k_{3} as arguments and finds and returns any solutions (as a numeric vector) in such a situation. This is achieved as follows:

- Evaluate k_{2}^{2} - 4k_{1}k_{3}. If this is *negative*, there are no solutions, and an appropriate message should be printed to the console.
- If k_{2}^{2} - 4k_{1}k_{3} is *zero*, then there is one solution, computed by (-k_{2}/2k_{}_{1}).
- If k_{2}^{2} - 4k_{1}k_{3} is *positive*, then there are two solutions:

$$(-k_2 - (k_2^2 - 4k_1k_3)^{0.5})/2k_1$$

$$(-k_2 + (k_2^2 - 4k_1k_3)^{0.5})/2k_1$$

- No default values are needed for the three arguments, but the function should check to see whether any are missing. If so, an appropriate character string message should be returned to the user, informing the user that the calculations are not possible.

Test your function:

1. Confirm the following:
   (a) (2x^{2} - x - 5) has roots 1.850781 and -1.350781
   (b) (x^{2} + x - 5) has no real roots
2. Attempt to find solutions to the following quadratic equations:
   (a) (1.3x^{2} - 8x - 3.13)
   (b) (2.25x^{2}^{} - 3x + 1)
   (c) (1.4x^{2} - 2.2x - 5.1)
   (d) (-5x^2 + 10.11x -9.9)
3. Test your programmed response in the function if one of the arguments is missing.

References

Argument Clinic. URL: wikipedia.org. Complete sketch on dailymotion.