Functions

Yufeng Huang

Associate Professor of Marketing, Simon Business School

August 15, 2022

Samsung's smartphone quantity choice

- Samsung sells a generic smartphone and needs to determine how many units to produce
  - for now, think of Samsung as the only player in this market
  - "Samsung vs HTC" gives a richer case where HTC is a competing player
- Production cost is \$100 per unit
- Market determines price, with

$$P(q) = 500 - 0.6 \cdot q$$

- Question: how many units does Samsung produce maximize its profits?

Samsung's smartphone quantity choice

- Could hand-calculate this problem for a few number of quantities:
  - q = 100, 200, 300 and 400
  - for the above quantities, market prices are $440, $380, $320, and $260
  - price minus costs are $340, $280, $220, and $160
  - profits are $34,000, $56,000, $66,000, and $64,000
- $\Rightarrow$ should produce 300 units

In code

```
# marginal costs
mc_S <- 100

# the four quantities
pos_qty <- c(100, 200, 300, 400)

# initialize the profit vector
profit_S <- rep(NA, length(pos_qty))

# loop to calculate the profits
for (q in 1:length(pos_qty)) {
        q_S <- pos_qty[q]
        price <- 500 - 0.6*q_S
        profit_S[q] <- q_S * (price - mc_S)
}

# examine results
data.frame(qty = pos_qty, profit = profit_S)

##   qty profit
## 1 100  34000
## 2 200  56000
## 3 300  66000
## 4 400  64000
```

What if market demand fluctuates?

- Market demand fluctuates: demand intercept is 200 in year 1, 300 in year 2, 400 in year 3

```
# Year 1:
profit_S_y1 <- rep(NA, length(pos_qty))

# loop to calculate the profits
for (q in 1:length(pos_qty)) {
        q_S <- pos_qty[q]
        price <- 200 - 0.6*q_S  # note the intercept is 200
        profit_S_y1[q] <- q_S * (price - mc_S)
}

# examine results
data.frame(qty = pos_qty, profit = profit_S_y1)

##   qty profit
## 1 100   4000
## 2 200  -4000
## 3 300 -24000
## 4 400 -56000
```

```r
# Year 3:
profit_S_y3 <- rep(NA, length(pos_qty))

# loop to calculate the profits
# you might see that we don't really need a loop to calculate profits,
# but we'll later extend this point to a separate point
for (q in 1:length(pos_qty)) {
        q_S <- pos_qty[q]
        price <- 400 - 0.6*q_S  # note the intercept is 200
        profit_S_y3[q] <- q_S * (price - mc_S)
}

# examine results
data.frame(qty = pos_qty, profit = profit_S_y3)

##   qty profit
## 1 100  24000
## 2 200  36000
## 3 300  36000
## 4 400  24000
```

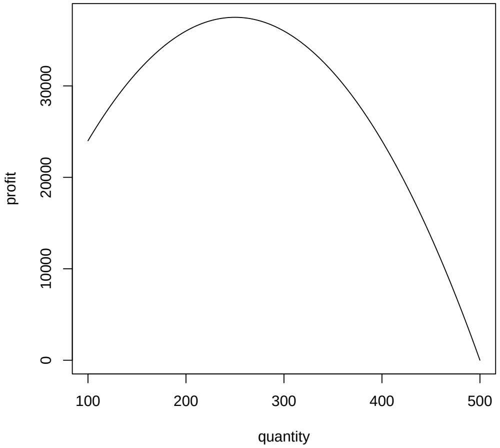
Anternative: write a function

```
# define a profit function
samsung_profit_vec <- function(intercept) {
        profit_S <- rep(NA, length(pos_qty))

        # loop to calculate the profits
        for (q in 1:length(pos_qty)) {
                q_S <- pos_qty[q]
                price <- intercept - 0.6*q_S    # note the intercept is an argument now
                profit_S[q] <- q_S * (price - mc_S)
        }

        # return
        return(profit_S)
}

# year 1
samsung_profit_vec(intercept = 200)     # can omit "intercept = "

## [1]   4000  -4000 -24000 -56000

# year 2
samsung_profit_vec(intercept = 300)

## [1]  14000  16000   6000 -16000

# year 3
samsung_profit_vec(intercept = 400)

## [1] 24000 36000 36000 24000
```

Alternative con'd: write a more flexible function if needed...

```
# define a profit function (more flexible)
samsung_profit_vec_2 <- function(intercept, pos_qty) {
        profit_S <- rep(NA, length(pos_qty))    # note: pos_qty is now an argument!

        # loop to calculate the profits
        for (q in 1:length(pos_qty)) {
                q_S <- pos_qty[q]
                price <- intercept - 0.6*q_S
                profit_S[q] <- q_S * (price - mc_S)
        }

        # return
        return(profit_S)
}
```

```r
# ... so that we can plot profit at a range of different quantity vectors
plot(100:500, samsung_profit_vec_2(intercept = 400, pos_qty = 100:500),
        type = 'l', xlab = "quantity", ylab = "profit")
```



So, why do we need to write functions?

- We need to repeatedly call a routine
  - the routine *itself* is systematic
  - but not systematic for:
    - when we need it
    - how we use it
  - and more importantly there are no existing functions that achieve what we want to do
- In the toy example
  - how to compute Samsung's profit is the systematic rule
  - where do we apply the rule is more flexible (or *should be* more flexible??)
  - e.g., we can later write a function about Samsung's profit under any quantity q_S, instead of the vector of Samsung's profit under a given set of quantities

Anatomy of a function

Anatomy of a function

- A function has a name assigned to it
- Takes zero, one or several inputs known as *arguments*
- The expressions forming the operations comprise the body of the function

```
# built-in examples
getwd() # no argument
mean(x, na.rm = T)      # one or two argument
plot(y ~ x, data, type = 'l', lty = 2)  # many arguments
```

- Returns a *single object*

Example: square

Write a function that squares its argument:

```
square <- function(x) {
        return(x^2)
}
```

- the function name is "square"
- the function has one argument
  - note: square(x) and square(y) are **calls** of the **same function**
- the function body consists of one simple expression "x^2"
- it returns the value "x^2"

Example: square

```
# works just like built-in functions in R
square(10)

## [1] 100

# works on vector too
square(1:5)

## [1]  1  4  9 16 25

#  question for you: why??

# returns error when we put characters on it
square("Hi I'm Yufeng")

## Error in x^2:  non-numeric argument to binary operator

#  question for you: why??
```

Defining a function[1]

The following are equivalent:

```r
# standard
square <- function(x) {
        return(x^2)
}

# can skip return()
square <- function(x) {
        x^2
}

# can write simple expression in one line
square <- function(x) x^2

# [EASILY MISREAD!] can even split into multiple lines
square <- function(x)
        x^2
```

[1] The last approach is very hard to read and easy to make mistake...

Side: Function name

- Cannot start with number of symbols or contain illegal characters

```
# all 3 are illegal
_square <- function(x) x^2

2power <- function(x) x^2

power-two <- function(x) x^2
```

- Can be defined as an operator
  - not adviced for beginners because 1) harder to read and 2) can override built-in operators

```
# define the reverse of power operator
`%^%` <- function(x, y) y^x
3%^%2

## [1] 8

# much better for readability is
reverse_power <- function(power, base) base^power
# why? 1) base and power are more concrete to understand
#    2) function name tells the reader what it does
```

Only allow one output (“multiple outputs” => group as a list)

```r
# square of sum: a function with two arguments
square_of_sum <- function(x, y) {
        return((x + y)^2)
}
square_of_sum(1, 2)
```

```
## [1] 9
```

```r
# can ONLY return a single object
square_of_sum <- function(x, y) {
        x_y <- x + y
        x_y_2 <- x_y^2
        return(list(sum = x_y, sum_square = x_y_2))
}
square_of_sum(1, 2)$sum # first element in the list
```

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

```r
square_of_sum(1, 2)$sum_square  # second element
```

```
## [1] 9
```

Arguments in a function

Can admit one or multiple arguments

```
# function with one argument
square(2)

## [1] 4

# function with two arguments
reverse_power <- function(power, base) base^power
reverse_power(3, 2)

## [1] 8

# function with multiple arguments
c(2, 1, 5, 2, 3, 6)

## [1] 2 1 5 2 3 6
```

Argument matching: can change order or arguments if they are clearly specified

```
# But, I don't quite remember what is what in reverse_power()

# can reverse argument explicitly
reverse_power(base = 2, power = 3)

## [1] 8

# cannot do so if we do not explicilty state input
reverse_power(2, 3)

## [1] 9
```

Argument matching: useful with functions with many (optional) arguments

```r
# aggregate example
res_1 <- aggregate(x = mpg ~ cyl, data = mtcars, FUN = mean)

# I don't have to remember the order of stuff...
res_2 <- aggregate(data = mtcars, FUN = mean, x = mpg ~ cyl)

# Can partially match argument names (partial matching)
res_3 <- aggregate(mtcars, FUN = mean, mpg ~ cyl)

# and gives the identical result
identical(res_1, res_2)

## [1] TRUE

identical(res_2, res_3)

## [1] TRUE
```

Can admit no argument

```r
# function with no argument
two_squared <- function() {
        x <- 2
        return(x^2)
}

# function with strictly no argument
two_squared()

## [1] 4

# returns error when you give it an argument (because it allows no argument)
two_squared(2)

## Error in two_squared(2):  unused argument (2)
```

Arguments *can* have default values[2]

```r
# function with default argument
squared <- function(x = 10) {
       return(x^2)
}

squared(2)

## [1] 4

squared()       # now gives default value x = 10

## [1] 100
```

[2] And this is different from no argument

Why have default values?

Default value is also very useful for functions with many arguments[3]

```
# plot, setting marker type as "points",
# color as "black", and marker size as 1
plot(mtcars$mpg, mtcars$wt, type = "p", col = "black", cex = 1)

# I could just leave everything to default
plot(mtcars$mpg, mtcars$wt)
```

[3] Results are the same but R recorded them a bit differently based on how some options are set

Can specify missing() to allow for missing arguments with no default

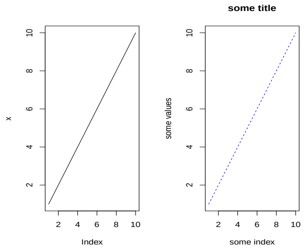
```
# sum over 3 numbers, one with default and one without
sum3 <- function(x, y, z = 0) {
        if (missing(y)) {
                res <- x + z
        } else {
                res <- x + y + z
        }
        return(res)
}

sum3(1)

## [1] 1

sum3(1, 2)

## [1] 3

sum3(1, 2, 3)

## [1] 6
```

Default value can depend on other variables

```
# default value as a function
x_plus_y <- function(x, y = log(x)) {
        return(x + y)
}

x_plus_y(2, 3)

## [1] 5

x_plus_y(2)     # y is missing! now take default y = log(x)

## [1] 2.693147

# WAIT: how did we know y = log(2)?
```

Can use dots (...) to pass more arguments to inside of a function

```
# Let's say we want to plot lines
lineplot <- function(x, ...) {
        plot(x, type = 'l', ...)
}

par(mfrow = c(1, 2))
# we can just plot a line
lineplot(1:10)
# or add other options
lineplot(1:10, xlab = "some index", ylab = "some values", main = "some title", col = "blue", lty = 2)
```




```
# but changing type will run into argument conflicts
lineplot(1:10, type = "p")

## Error in plot.default(x, type = "l", ...):  formal argument "type" matched by multiple actual
arguments
```

Built-in example

paste {base} R Documentation

Concatenate Strings

Description

Concatenate vectors after converting to character.

Usage

```
paste (..., sep = " ", collapse = NULL)
paste0(..., collapse = NULL)
```

Arguments

`...`	one or more R objects, to be converted to character vectors.
`sep`	a character string to separate the terms. Not `NA_character_`.
`collapse`	an optional character string to separate the results. Not `NA_character_`.

```
# paste()
paste("We", "are", "awesome")   # default for argument sep is " "

paste("We", "are", "awesome", "R", "Programmers")       # varies the number of arguments, still works

paste("We", "are", sep = " ", "awesome")        # argument matching; but this is confusing...

## [1] "We are awesome"
## [1] "We are awesome R Programmers"
## [1] "We are awesome"
```

Side: Operators as functions

Operators as functions

```
# `` is the notation to refer to an operator
`+`(2, 3)       # equivalent to 2 + 3

## [1] 5

`+`(2, `*`(2, 3))

## [1] 8

2 + (2 * 3)     # equivalent

## [1] 8

# define an array
my.vec <- c("one", "two", "three", "four")      # 2 by 2 by 2 cube
`[`(my.vec, 2)

## [1] "two"

my.vec[2]       # equivalent

## [1] "two"
```

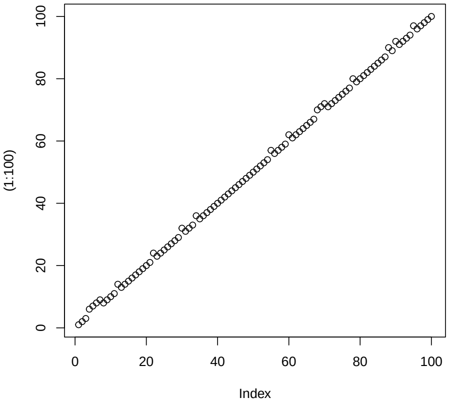
A more advanced example: define your own operators[4]

```r
# Courtecy of Hadley Wickham, let's do a prank
#  runif() generates random number between 0 and 1
#  evenly distributed so 15% of runif(length(x)) will fall below 0.15

# now let's define a function `(` -- as the parenthesis operator
`(` <- function(x) {
        ifelse(
                runif(length(x)) < 0.15, # condition
                x + 2,  # if TRUE
                x       # if FALSE
        )
}
```

[4] Strictly speaking, runif() funds uniform random numbers

A more advanced example: define your own operators

```
# now, plotting (1:100) will not work because `(` has effect on 1:100
plot((1:100))   # note that plot(1:100) does not change
```

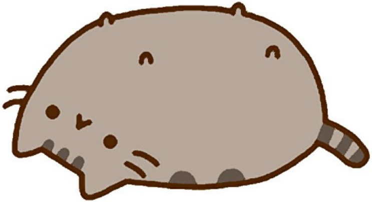



```
# remove this to be safe
rm(`(`)
```

What we know and what we don't know

- What we know
  - fun_name <- function(x) { body }
  - argument and argument matching
- What we don't know
  - lazy evaluation
  - environment and scoping
- And importantly: more applications

R is lazy



Lazy evaluation 1: only needs an argument when actually evaluating it[5]

```r
# Consider a function with arguments a and b
g <- function(a, b) {
        return(a * a * a)
}

# call g(2) will be equivalent to calling g(2, 1)
g(2, 1) # standard call, but b is useless
```

```
## [1] 8
```

```r
g(2)    # ignores b
```

```
## [1] 8
```

[5] Lasy Evaluation IS UNIQUE in R!!

Lazy evaluation 2: runs into error when – not before – calling arguments that do not exist

```
# Consider a function with arguments a and b
g <- function(a, b) {
        print(a * a * a)
        print(b)
}

# now g(2) will be different from g(2, 1)
g(2, 1) # standard call, now b is useful

## [1] 8
## [1] 1

g(2)    # still evaluates print(a^3)

## [1] 8

     ## Error in print(b):  argument "b" is missing, with no default
```

Lazy evaluation 3: default arguments evaluated only when called upon

```
# Consider a function with arguments a and b
g <- function(a, b = 3*a) {
        print(a * a * a)
        a <- a^3
        print(b)
}

# now g(2, 1) takes standard values given by arguments
g(2, 1)

## [1] 8
## [1] 1

# but g(2) uses b that comes from the UPDATED a
g(2)    # a updated to 2^3, then default value of b is 3*8

## [1] 8
## [1] 24
```

Environments and variable scoping

Motivating example: what is x?

```
# What is x after the following?

rm('x') # remove any x in the memory

              ## Warning in rm("x"):  object 'x' not found

f <- function(x) {
        x <- x^2
        return(x)
        x^2
}
f(2)

## [1] 4

# NOW WHAT IS x?
```

```
x

## Error in eval(expr, envir, enclos):  object 'x' not found

# x does not exist
# WHY?
```

Variables in local vs global environments

```
# How does this structure work?
y <- 10
f <- function(x) {
        return(x + y)
}
f(0)

## [1] 10

# How does this different structure work?
y <- 10
g <- function(x) {
        y <- 5
        return(x + y)
}
g(0)

## [1] 5

# After running these lines, what's y?
y

## [1] 10
```

```r
# Can print the y variables in given environment to see what happens...

# How does this structure work?
y <- 10
f <- function(x) {
        print(paste("y is", y)) # local y
        return(x + y)
}
f(0)
```

```
## [1] "y is 10"
## [1] 10
```

```r
# How does this different structure work?
y <- 10
g <- function(x) {
        y <- 5
        print(paste("y is", y)) # local y
        return(x + y)
}
g(0)
```

```
## [1] "y is 5"
## [1] 5
```

```r
# After running these lines, what's y?
print(paste("y is", y)) # global y
```

```
## [1] "y is 10"
```

What happened?

- In the first example, can't find local y so search the global environment
- In the second example, y is defined in local environment so use local y
- While running g(), global y is not changed, so y $=$ 5 is only for inside g()
- Therefore, after running g(), y in the global environment remains unchanged

Scope of a variable

```
# The scope of a variable is the range of places where you can find it
# R will first try to find the variable in the current environment
y <- 10
paste("y in global is", y)      # global

## [1] "y in global is 10"

f <- function() {
        y <- 5
        print(paste("y in f() is", y))
}
f()     # local to function f

## [1] "y in f() is 5"
```

Scope of a variable

```
# The scope of a variable is the range of places where you can find it
# R will first try to find the variable in the current environment

# If there is no such variable y, R will go up and find in parent environment
f <- function() {
        g <- function() {
                print(paste("y in g() is", y))
                # no y in g, search in environment of f
        }
        print(paste("y in f() is", y))  # no y; search global environment
        y <- 5
        return(g())
}
f()     # first the global y and then the local-to-f() y

## [1] "y in f() is 10"
## [1] "y in g() is 5"
```

Your turn: scope of a variable

```
# clear all previous memory
rm(list = ls())

# g() finds y in the environment of f()
f <- function(x) {
        y <- 10
        g <- function(x) {
                x + y
        }
        return(g(x))
}
f(5)

## [1] 15

# how about this?
f <- function(x) {
        y <- 10
        return(g(x))
}
g <- function(x) {
        return(x + y)
}
# f(5) is ?
```

What happened?

- In the first example, g() is defined in the environment of f()
  - f() passes x to g()
  - g() needs y, so searches the parent environment (i.e., that of f()) to find y
  - y is defined in the environment of f()
- In the second example, f() is defined in the global environment
  - f() calls g() inside, can't find g() inside its own environment
  - finds g() in the parent environment, so global
  - g() is defined in global so calls y in global
  - can't find it, returns error

A more advanced example

```
# clear all previous memory
rm(list = ls())

# we first assign a function of which the OUTPUT IS A FUNCTION
j <- function(x) {
        y <- 2
        return(function() c(x, y))      # returns this function
}
k <- j(1)       # so k is a function

# side: this is different from
# j <- function(x) {
#     y <- 2
#     return(c(x, y))
# }

# note that k maintains the environment within j
#    so even call k in global
k()
```

```
## [1] 1 2
```

```
#    it gives the y value in j()
```

Examples / exercises

Example 1: check positive value

```
# Let's write a function that checks positive value
check_positive <- function(x) {
        if (x > 0) print(TRUE) else print(FALSE)
}

# instead of printing some value, more often we "return" the value
check_positive <- function(x) {
        return(x > 0)
}
```

Example 1: check positive value

```r
# simple check_positive
check_positive <- function(x) {
        return(x > 0)
}

# Your turn: output?
check_positive(0)

check_positive(NA)

check_positive(TRUE)

check_positive("positive")

check_positive(-2:2)
```

Example 2: root of quadratic equations[6]

- Suppose I want to evaluate the root x of quadratic equations with the general form

$$a \cdot x^2 + b \cdot x + c = 0$$

where a, b and c are parameters

- Mathematically there are two solutions of x, jointly expressed as

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

[6] Which we talked about in Week 1

Example 2: root of quadratic equations

```
quadratic_root <- function(a, b, c) {
    sqr_root <- sqrt(b^2 - 4*a*c)
    x1 <- (-b + sqr_root) / (2*a)
    x2 <- (-b - sqr_root) / (2*a)
    return(list(root1 = x1, root2 = x2))
}

# root of -x^2 + 2x + 1 = 0
x <- quadratic_root(-1, 2, 1)
x$root1
```

```
## [1] -0.4142136
```

```
x$root2
```

```
## [1] 2.414214
```

```r
# root of x^2 + x + 1 = 0
x <- quadratic_root(1, 1, 1)

## Warning in sqrt(b^2 - 4 * a * c):  NaNs produced

# warning and no result in real number (returns NaN)
x$root1

## [1] NaN
```

```r
# instead, wrap around a warning message and return complex number
quadratic_root <- function(a, b, c) {
        if (b^2 - 4*a*c >= 0) {
                sqr_root <- sqrt(b^2 - 4*a*c)
        } else {
                warning("equation has no real root, complex root returned")
                sqr_root <- sqrt(- b^2 + 4*a*c)*1i
        }
        x1 <- (-b + sqr_root) / (2*a)
        x2 <- (-b - sqr_root) / (2*a)
    return(list(x1, x2))
}
quadratic_root(1, 1, 1)
```

```
## Warning in quadratic_root(1, 1, 1):  equation has no real root,
complex root returned
```

```
## [[1]]
## [1] -0.5+0.8660254i
##
## [[2]]
## [1] -0.5-0.8660254i
```

The warning() and stop() functions produce warning and error messages

```
gen_error <- function(x) {
        stop("hey this is an error code and function stops here")
        return(x^2)
}
gen_error(2)

## Error in gen_error(2):  hey this is an error code and function stops
here

gen_warning <- function(x) {
        warning("hey this is a warning message and the code goes on")
        return(x^2)
}
gen_warning(2)  # does not stop the code!

## Warning in gen_warning(2):  hey this is a warning message and the
code goes on

## [1] 4
```

Example 3 and optimization

Example 3: profit maximization

- Recall: Samsung wants to maximize profit, which depends on its quantity, demand level (how good the market is), and marginal costs

$$\Pi(q) = q \times (P(q) - c)$$

- Inverse demand is

$$P(q) = \text{demand} - 0.6 \times q$$

- We want to know how to set prices that maximizes Π_t given the demand intercept ("demand") and cost

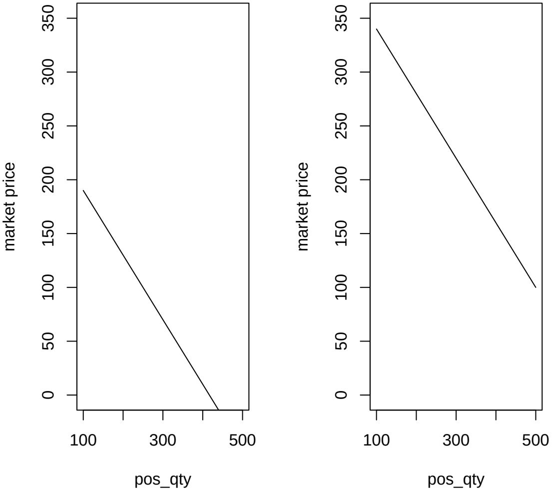
Let's write the profit function

```
# demand function
#    note: careful to include variables as arguments (prevent scoping)
market_price <- function(quantity, intercept) {
        price <- intercept - 0.6 * quantity
        return(price)
}

# profit function
profit <- function(quantity, intercept, cost) {
        price <- market_price(quantity, intercept)
        profit <- quantity*(price - cost)
        return(profit)
}
```

In other words, profit() is a function that nests the function market_price()

Why nesting functions?

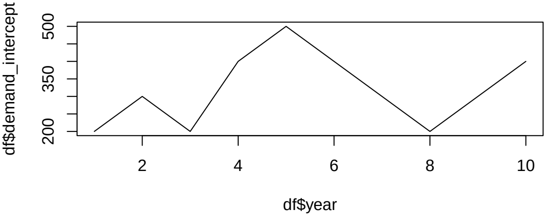
```r
# why nest function? Mostly because
#    1) I need market_price() somewhere else or
#    2) I need to call market_price() multiple times (e.g. this plot)
pos_qty <- 100:500        # possible quantities on the x-axis
par(mfrow = c(1, 2))    # allocate left/right graph

# demand under different market conditions
for (incpt in c(250, 400)) {
        plot(pos_qty, market_price(pos_qty, incpt),
                type = 'l', ylim = c(0, 350), ylab = "market price")
}
```

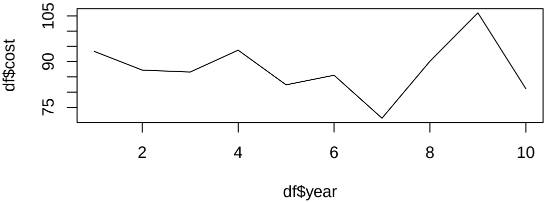

market price
350
300
250
200
150
100
50
0
100
300
500
pos_qty
market price
350
300
250
200
150
100
50
0
100
300
500
pos_qty

Now let's optimize Samsung's quantity across different demand conditions

```
##    year demand_intercept    cost
## 1     1              200  93.357
## 2     2              300  87.213
## 3     3              200  86.581
## 4     4              400  93.752
## 5     5              500  82.383
## 6     6              400  85.532
## 7     7              300  71.438
## 8     8              200  90.128
## 9     9              300 105.996
## 10   10              400  81.100
```

Demand and costs in 10 years

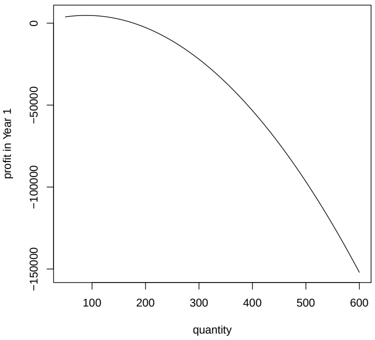




How would we maximize profit? Brute-force approach

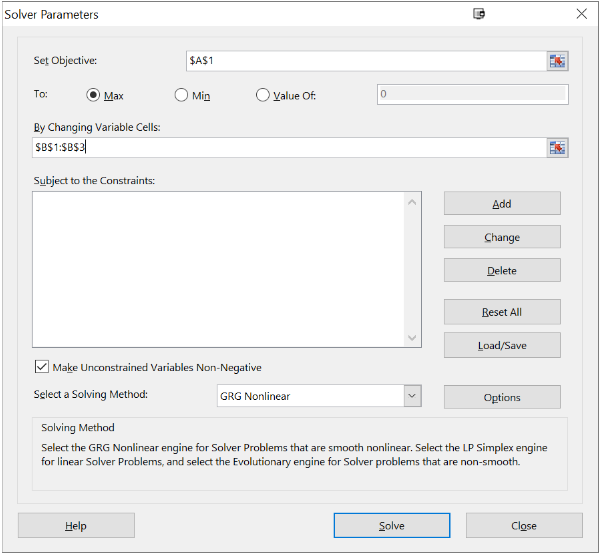
- A brute-force approach is to solve for all profits under all possible quantities
- Question: what's the problem with this approach?

```r
# brute-force approach: take all possible quantities for each year
pos_qty <- 50:600

# example: year 1
profit_vec_1 <- profit(pos_qty, df$demand_intercept[1], df$cost[1])

# plot
plot(pos_qty, profit_vec_1,
        xlab = "quantity", ylab = "profit in Year 1", type = 'l')
```




```r
# which quantity solves the highest profit?
pos_qty[which.max(profit_vec_1)]

## [1] 89
```

How would we maximize profit if this is Excel?



A better approach: optim()

```
# optim() is like an Excel solver, but in R (and much better than Excel)
optimization_result <- optim(
        par = 100,      # initial trial value
        fn = function(qty) {
                - profit(qty, df$demand_intercept[1], df$cost[1])
        }       # function with ONE ARGUMENT to MINIMIZE
)

# returns a list of results
optimization_result[1:2]

# first is quantity, second is -profit

## $par
## [1] 88.86719
##
## $value
## [1] -4738.68
```

Repeat this for many years

```
# for loop to wrap this optimization routine
df$optimal_quantity <- rep(NA, 10)
for (i in 1:nrow(df)) {
        df$optimal_quantity[i] <- optim(par = 100, function(qty) {
                - profit(qty, df$demand_intercept[i], df$cost[i])
                }       # the function will look for i outside of its body
        )[[1]]  # take first element which is profit-max quantity
}
df$market_price <- market_price(df$optimal_quantity, df$demand_intercept)

head(round(df, 3), n = 10)

##    year demand_intercept    cost optimal_quantity market_price
## 1     1              200  93.357           88.867      146.680
## 2     2              300  87.213          177.305      193.617
## 3     3              200  86.581           94.512      143.293
## 4     4              400  93.752          255.215      246.871
## 5     5              500  82.383          348.008      291.195
## 6     6              400  85.532          262.070      242.758
## 7     7              300  71.438          190.312      185.812
## 8     8              200  90.128           91.562      145.062
## 9     9              300 105.996          161.660      203.004
## 10   10              400  81.100          265.742      240.555
```

Summary

- **Assigning** a function
  - function_name <- function(argument1, argument2) { body_of_the_function }
- **Calling** a function
  - function_name(argument1, argument2)
- Properties of the function (important things about arguments)
  - "..." passes arguments
  - argument matching
  - lazy evaluation
  - variable scoping
- optim()