COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

GR5206: lecture 2

Computational Statistics
And Introduction to Data Science

Thibault Vatter
Department of Statistics, Columbia University

9/13/2019

Outline

1 Control flow

2 Choices

3 Loops

4 Functions

Control flow

- Two primary tools of control flow:
  - Choices.
  - Loops.
- Choices:
  - E.g. `if()`, `ifelse()`, `switch()`.
  - Allows to run different code depending on the input.
- Loops:
  - E.g. `for`, `while`, `repeat`.
  - Allows to repeatedly run code, typically with changing options.
- You might want to have a look at chapter 8 for condition system (messages, warnings, and errors)...

Outline

1 Control flow

2 Choices

3 Loops

4 Functions

if() statements

- The basic idea for if statements:
  - If condition is TRUE, true_action is evaluated.
  - If condition is FALSE, the optional false_action is evaluated.

```
if (condition) true_action
if (condition) true_action else false_action
```

- Typically, actions are compound statements contained within {.

```
grade <- function(x) {
  if (x > 90) {
    "A"
  } else if (x > 80) {
    "B"
  } else if (x > 50) {
    "C"
  } else {
    "F"
  }
}
```

if() statements cont'd

- if returns a value so that you can assign the results:
  - Only do that when it fits on one line; otherwise hard to read.

```
x1 <- if (TRUE) 1 else 2
x2 <- if (FALSE) 1 else 2

c(x1, x2)
#> [1] 1 2
```

- When using if without else:
  - Returns NULL if the condition is FALSE.
  - Useful with functions like c()/paste() dropping NULL inputs.

```
greet <- function(name, birthday = FALSE) {
  paste0("Hi ", name, if (birthday) " and HAPPY BIRTHDAY")
}
greet("Maria", FALSE)
#> [1] "Hi Maria"
greet("Jaime", TRUE)
#> [1] "Hi Jaime and HAPPY BIRTHDAY"
```

Invalid inputs

- The condition should evaluate to a single `TRUE` or `FALSE`:

```
if ("x") 1
#> Error in if ("x") 1: argument is not interpretable as logical
if (logical()) 1
#> Error in if (logical()) 1: argument is of length zero
if (NA) 1
#> Error in if (NA) 1: missing value where TRUE/FALSE needed
```

- The exception (frequent source of bugs, avoid):
  - A logical vector of length greater than 1 generates a warning.

```
if (c(TRUE, FALSE)) 1
#> Warning in if (c(TRUE, FALSE)) 1: the condition has length > 1 and
#> only the first element will be used
#> [1] 1
```

- In R >=3.5.0+, you can turn this into an error (good practice):

```
Sys.setenv("_R_CHECK_LENGTH_1_CONDITION_" = "true")
if (c(TRUE, FALSE)) 1
#> Error in if (c(TRUE, FALSE)) 1: the condition has length > 1
```

Vectorised `if()` statements

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

- `if` only works with a single `TRUE` or `FALSE`.
- What if you have a vector of logical values?
- Answer: `ifelse()`, a vectorised function with `test`, `yes`, and `no` vectors (recycled to the same length).
  - Missing values are propagated into the output.
  - Advice: use `ifelse()` only when the `yes` and `no` vectors (otherwise hard to predict the output type).

```
x <- 1:10
ifelse(x %% 5 == 0, "XXX", as.character(x))
#>  [1] "1"   "2"   "3"   "4"   "XXX" "6"   "7"   "8"   "9"   "XXX"

ifelse(x %% 2 == 0, "even", "odd")
#>  [1] "odd"  "even" "odd"  "even" "odd"  "even" "odd"  "even" "odd"
#> [10] "even"
```

switch() statements

- Lets you replace code like:

```
x_option <- function(x) {
  if (x == "a") {
    "option 1"
  } else if (x == "b") {
    "option 2"
  } else {
    stop("Invalid `x` value")
  }
}
```

- With:

```
x_option <- function(x) {
  switch(x,
    a = "option 1",
    b = "option 2",
    stop("Invalid `x` value")
  )
}
```

switch() statements cont'd

- Last component should always throw an error.

```r
(switch("c", a = 1, b = 2))
#> NULL
```

- When multiple inputs share an output:
  - Use empty right hand sides of =.
  - Same as C's `switch` statement.

```r
legs <- function(x) {
  switch(x,
    cow = ,
    horse = ,
    dog = 4,
    human = ,
    chicken = 2,
    plant = 0,
    stop("Unknown input")
  )
}
legs("cow")
#> [1] 4
legs("dog")
#> [1] 4
```

- `switch()` with a numeric `x` is not recommended.

Outline

1 Control flow

2 Choices

3 Loops

4 Functions

Loops

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

- `for` loops are used to iterate over items in a vector.

```
for (item in vector) perform_action
```

- For each item in `vector`, `perform_action` is called once; updating the value of `item` each time.

```
for (i in 1:3) {
  print(i)
}
#> [1] 1
#> [1] 2
#> [1] 3
```

- When iterating over indices, use very short variable names like `i`, `j`, or `k` by convention.
- Important: `for` assigns the `item` to the current environment.

```
i <- 100
for (i in 1:3) {}
i
#> [1] 3
```

Early termination

- Two ways to terminate a `for` loop early:
  - `next` exits the current iteration.
  - `break` exits the entire `for` loop.

```r
for (i in 1:10) {
  if (i < 3)
    next

  print(i)

  if (i >= 5)
    break
}
#> [1] 3
#> [1] 4
#> [1] 5
```

Common pitfalls

- Three common pitfalls to watch out for when using `for`:
  - Preallocation.
  - Iteration over e.g. `1:length(x)`.
  - Iteration over S3 vectors.
- Preallocation:
  - If you're generating data, preallocate the output.
  - Otherwise the loop will be very slow.
  - `vector()` function is helpful.

```
means <- c(1, 50, 20)
out <- vector("list", length(means))
for (i in 1:length(means)) {
  out[[i]] <- rnorm(10, means[[i]])
}
```

Common pitfalls cont'd

- Next, beware of iterating over `1:length(x)`, which will fail in unhelpful ways if `x` has length 0.

```
means <- c()
out <- vector("list", length(means))
for (i in 1:length(means)) {
  out[[i]] <- rnorm(10, means[[i]])
}
#> Error in rnorm(10, means[[i]]): invalid arguments

# The reason? `:` works with both increasing and decreasing sequences.
1:length(means)
#> [1] 1 0
```

- Use `seq_along(x)` instead.

```
seq_along(means)
#> integer(0)

out <- vector("list", length(means))
for (i in seq_along(means)) {
  out[[i]] <- rnorm(10, means[[i]])
}
```

Common pitfalls cont'd

- Finally, problems arise when iterating over S3 vectors, as loops typically strip the attributes.

```
xs <- as.Date(c("2020-01-01", "2010-01-01"))
for (x in xs) {
  print(x)
}
#> [1] 18262
#> [1] 14610
```

- Work around this by using `[[`.

```
for (i in seq_along(xs)) {
  print(xs[[i]])
}
#> [1] "2020-01-01"
#> [1] "2010-01-01"
```

Related tools

- `for` loops:
  - Useful when known in advance the set of values to iterate over.
  - Otherwise:
    - `while(condition) action`: performs `action` while `condition` is `TRUE`.
    - `repeat(action)`: repeats `action` forever (i.e. until it encounters `break`).
    - Possible to write any `for` using `while`, and any `while` using `repeat`, but not the converse.
    - Good practice to use the least-flexible (i.e., simplest) solution to a problem.
- R does not have a `do {action} while (condition)` syntax found in other languages.
- Generally speaking you shouldn't need to use `for` loops for **data analysis tasks**, we'll see better solutions.

Outline

1 Control flow

2 Choices

3 Loops

4 Functions

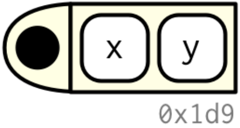
Functions

- Two **important ideas**:
  - Functions can be broken down into three components: arguments, body, and environment.
  - Functions are objects, just as vectors are objects.
- In the following:
  - **The basics:** how to create functions and the three main components of a function.
  - **Function composition:** the three forms commonly used in R code.
  - **Lexical scoping:** how R finds the value associated with a given name.
  - **Lazy evaluation:** the fact that function arguments are only evaluated when used for the first time.
  - **The special ... argument:** how to pass on extra arguments to another function.
  - **Exiting a function:** how can a function exit and exit handlers.

Function components

- A function has three parts:
  - The `formals()`, list of arguments controling how you call the function.
  - The `body()`, the code inside the function.
  - The `environment()`, the data structure that determines how the function finds the values associated with the names.

```r
f02 <- function(x, y) {
  # A comment
  x + y
}
```



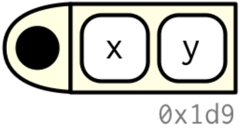
Function components cont'd

- How those are defined?
  - Explicitly for the formals and body.
  - Implicitly for the environment (where the function was defined).

```
formals(f02)
#> $x
#>
#>
#> $y

body(f02)
#> {
#>     x + y
#> }

environment(f02)
#> <environment: R_GlobalEnv>
```



Function components cont'd

- Functions can possess any number of additional `attributes()`.
- One attribute in base R is `srcref`, short for source reference.
  - Points to the source code used to create the function.
  - Used for printing because, unlike `body()`, it contains code comments and other formatting.

```
attr(f02, "srcref")
#> function(x, y) {
#>   # A comment
#>   x + y
#> }
```

Primitive functions

- One exception to the three components rule.
- Call C code directly.

```
sum
#> function (..., na.rm = FALSE)  .Primitive("sum")
`[`
#> .Primitive("[")
```

- Type is either `builtin` or `special`.

```
typeof(sum)
#> [1] "builtin"
typeof(`[`)
#> [1] "special"
```

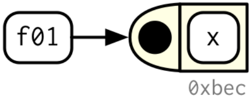
- `formals()`, `body()`, and `environment()` are all `NULL`.

```
formals(sum)
#> NULL
body(sum)
#> NULL
environment(sum)
#> NULL
```

First-class functions

- R functions are objects in their own right!
- This language property often called "first-class functions".
- Unlike in many other languages, no special syntax:
  - Create a function object (with `function`).
  - Bind it to a name with `<-`.

```r
f01 <- function(x) {
  sin(1 / x ^ 2)
}
```



More on functions

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

- The binding step is not compulsory.
- A function without a name is called an **anonymous function**.

```
lapply(mtcars, function(x) length(unique(x)))
integrate(function(x) sin(x) ^ 2, 0, pi)
```

- Also possible to put functions in a list.

```
funs <- list(
  half = function(x) x / 2,
  double = function(x) x * 2
)

funs$double(10)
#> [1] 20
```

- In R, functions are often called **closures**.
- The name reflects the fact that R functions capture/enclose, their environments.

Invoking a function

Columbia University in the City of New York

- The standard way:

```r
mean(1:10, na.rm = TRUE)
#> [1] 5.5
```

- The alternative way:

```r
args <- list(1:10, na.rm = TRUE)
do.call(mean, args)
#> [1] 5.5
```

Function composition

- Imagine you want to compute the population standard deviation using `sqrt()` and `mean()`.

```
square <- function(x) x^2
deviation <- function(x) x - mean(x)
```

- Either nest the function calls.

```
x <- runif(100)

sqrt(mean(square(deviation(x))))
#> [1] 0.278
```

- Or save the intermediate results as variables.

```
out <- deviation(x)
out <- square(out)
out <- mean(out)
out <- sqrt(out)
out
#> [1] 0.278
```

Function composition cont'd

- The third option using the `magrittr` package:
  - The operator %>%, called **pipe** and pronounced as "and then".

```r
library(magrittr)

x %>%
  deviation() %>%
  square() %>%
  mean() %>%
  sqrt()
#> [1] 0.278
```

- Advantages:
  - Focus on the high-level composition of functions, not the low-level flow of data.
  - Focus on what's being done (the verbs), not on what's being modified (the nouns).
  - Makes your code more readable by:
    - Structuring sequences of data operations left-to-right.
    - Minimizing the need for local variables and function definitions.
    - Making it easy to add steps anywhere in the sequence.

Basic piping

- `x %>% f` is equivalent to `f(x)`
- `x %>% f(y)` is equivalent to `f(x, y)`
- `x %>% f(y) %>% g(z)` is equivalent to `g(f(x, y), z)`

```r
x <- 1:10
y <- x + 1
z <- y + 1
f <- function(x, y) x + y

x %>% sum
#> [1] 55
x %>% f(y)
#>  [1]  3  5  7  9 11 13 15 17 19 21
x %>% f(y) %>% f(z)
#>  [1]  6  9 12 15 18 21 24 27 30 33
```

The argument ("dot") placeholder

- `x %>% f(y, .)` is equivalent to `f(y, x)`
- `x %>% f(y, z = .)` is equivalent to `f(y, z = x)`

```r
x <- 1:10
y <- 2 * x
f <- function(z, y) y / z

x %>% f(y, .)
#>  [1] 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5 0.5
x %>% f(y, z = .)
#>  [1] 2 2 2 2 2 2 2 2 2 2
```

Function composition cont'd

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

- Each of the three options has its own strengths and weaknesses:
  - Nesting, `f(g(x))`:
    - Concise, and well suited for short sequences.
    - Longer sequences harder to read (inside out & right to left).
    - Arguments can get spread out over long distances creating the Dagwood sandwich problem.
  - Intermediate objects, `y <- f(x); g(y)`:
    - Requires you to name intermediate objects.
    - A strength when objects are important, but a weakness when values are truly intermediate.
  - Piping, `x %>% f() %>% g()`:
    - Allows to read code in straightforward left-to-right fashion.
    - Doesn't require to name intermediate objects.
    - Only for linear sequences of transformations of a single object.
- Most code use a combination of all three styles, but...
- **Piping is more common in data analysis code!**

Lexical scoping

- **Scoping:** the act of finding the value associated with a name.
- What does the following code return?

```
x <- 10
g01 <- function() {
  x <- 20
  x
}

g01()
```

- R uses **lexical scoping**[1]:
  - Looks up the values of names based on how a function is defined, not how it is called.
  - Follows four primary rules:
    - Name masking
    - Functions versus variables
    - A fresh start
    - Dynamic lookup

[1] but possible to override the default rules.

Name masking

- Names defined inside a function mask names defined outside.

```r
x <- 10
y <- 20
g02 <- function() {
  x <- 1
  y <- 2
  c(x, y)
}
g02()
#> [1] 1 2
```

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

```r
x <- 2
g03 <- function() {
  y <- 1
  c(x, y)
}
g03()
#> [1] 2 1
y
#> [1] 20
```

Name masking cont'd

Columbia University in the City of New York

- Same applies if a function is defined inside another function:
  - First, R looks inside the current function.
  - Then, where that function was defined (and so on, all the way up to the global environment).
  - Finally, in other loaded packages.
- What does the following code return?

```
x <- 1
g04 <- function() {
  y <- 2
  i <- function() {
    z <- 3
    c(x, y, z)
  }
  i()
}
g04()
```

Functions versus variables

Columbia University in the City of New York

- Functions are ordinary objects, the same rules apply to them.

```
g07 <- function(x) x + 1
g08 <- function() {
  g07 <- function(x) x + 100
  g07(10)
}
g08()
#> [1] 110
```

- When a function and a non-function (in different environments) share the same name, the rules get a little more complicated.
  - When using a name in a function call, R ignores non-function objects when looking for that value.

```
g09 <- function(x) x + 100
g10 <- function() {
  g09 <- 10
  g09(g09)
}
g10()
#> [1] 110
```

- For the record, using the same name for different things is confusing and best avoided!

A fresh start

- What happens to values between invocations of a function?
- What will happen the first time you run this function?
- What will happen the second time?

```
g11 <- function() {
  if (!exists("a")) {
    a <- 1
  } else {
    a <- a + 1
  }
  a
}

g11()
g11()
```

Dynamic lookup

- The output of a function can depend on objects outside its environment, because:
  - Lexical scoping determines where, not when, to look for values.
  - R looks for values when the function is run, not when the function is created.

```r
g12 <- function() x + 1
x <- 15
g12()
#> [1] 16

x <- 20
g12()
#> [1] 21
```

- Can be quite annoying.
  - With spelling mistakes, no error when creating a function.
  - Depending on the global environment, maybe not even an error when running the function.

Lazy evaluation

- In R, function arguments are **lazily evaluated**:
  - Only evaluated if accessed.
  - What will this code return?

```
h01 <- function(x) {
  10
}
h01(stop("This is an error!"))
```

- Important because it allows to do things like include expensive computations in function arguments that are only be evaluated if needed.

Promises

- Lazy evaluation is powered by a data structure called a **promise**, or (less commonly) a thunk.
- A promise has three components:
  - An expression, like `x + y`, giving rise to the delayed computation.
  - An environment, where the expression should be evaluated.
  - A value.

Promises: the environment

- The environment is where the expression should be evaluated.
  - I.e., where the function is called.
  - What will this code return?

```
y <- 10
h02 <- function(x) {
  y <- 100
 x + 1
}

h02(y)
```

- Also means that when assignming inside a call to a function, the variable is bound outside the function, not inside.

```
h02(y <- 1000)
#> [1] 1001
y
#> [1] 1000
```

Promises: the value

- The value
  - Is computed and cached the first time a promise is accessed, when the expression is evaluated in the specified environment.
  - Ensures that the promise is evaluated at most once.
- What will this code return?

```r
double <- function(x) {
  message("Calculating...")
  x * 2
}

h03 <- function(x) {
  c(x, x)
}

h03(double(x))
```

- Can't manipulate promises with R code: any inspection attempt with code will force an immediate evaluation, making the promise disappear.

Default arguments

- Thanks to lazy evaluation:
  - Default values can be defined in terms of other arguments.
  - Or even in terms of variables defined later in the function.

```
h04 <- function(x = 1, y = x * 2, z = a + b) {
  a <- 10
  b <- 100

  c(x, y, z)
}

h04()
#> [1]   1   2 110
```

- Many base R functions use this technique, but not recommended:
  - Makes the code harder to understand.
  - To predict *what* will be returned, need to know the exact order in which default arguments are evaluated.

Default arguments

- The evaluation environment is slightly different for default and user supplied arguments.
  - User supplied arguments: evaluated in the global environment.
  - Default arguments: evaluated inside the function.
- Means that seemingly identical calls can yield different results.

```r
h05 <- function(x = ls()) {
  a <- 1
  x
}

# ls() evaluated in global environment:
h05(ls())
#> [1] "h05"

# ls() evaluated inside h05:
h05()
#> [1] "a" "x"
```

Missing arguments

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

- Use `missing()` to determine if an argument's value comes from the user or from a default.

```
h06 <- function(x = 10) {
  list(missing(x), x)
}
str(h06())
#> List of 2
#>  $ : logi TRUE
#>  $ : num 10
str(h06(10))
#> List of 2
#>  $ : logi FALSE
#>  $ : num 10
```

Missing arguments cont'd

COLUMBIA UNIVERSITY IN THE CITY OF NEW YORK

- How many arguments are required?

```r
args(sample)
#> function (x, size, replace = FALSE, prob = NULL)
#> NULL
```

- A "better"" `sample()`:
  - Use an explicit `NULL` to indicate that `size` is not required but can be supplied.

```r
sample <- function(x, size = NULL, replace = FALSE, prob = NULL) {
  if (is.null(size)) {
    size <- length(x)
  }

  x[sample.int(length(x), size, replace = replace, prob = prob)]
}
```

... (dot-dot-dot)

- The special argument `...` (pronounced dot-dot-dot).
  - Makes a function take any number of additional arguments.
  - In other programming languages:
    - This is often called *varargs* (short for variable arguments).
    - A function that uses it is said to be variadic.
- Can pass those additional arguments on to another function.

```
i01 <- function(y, z) {
  list(y = y, z = z)
}

i02 <- function(x, ...) {
  i01(...)
}

str(i02(x = 1, y = 2, z = 3))
#> List of 2
#>  $ y: num 2
#>  $ z: num 3
```

The two primary uses of . . .

- If a function takes a function as an argument, you want some way to pass additional arguments to that function.

```
x <- list(c(1, 3, NA), c(4, NA, 6))
str(lapply(x, mean, na.rm = TRUE))
#> List of 2
#>  $ : num 2
#>  $ : num 5
```

- If a function is an S3 generic, it needs some way to allow methods to take arbitrary extra arguments.

```
print(factor(letters), max.levels = 4)

print(y ~ x, showEnv = TRUE)
```

. . . cont'd

- `list(...)` evaluates the arguments and stores them in a list.

```
i04 <- function(...) {
  list(...)
}
str(i04(a = 1, b = 2))
#> List of 2
#>  $ a: num 1
#>  $ b: num 2
```

- In general, using `...` comes with two downsides:
  - When using it to pass arguments to another function, need to carefully explain to the user where those arguments go.
    - Makes it hard to understand what a function can do.
  - A misspelled argument will not raise an error.
    - Makes it easy for typos to go unnoticed.

```
sum(1, 2, NA, na_rm = TRUE)
#> [1] NA
```

Exiting a function

- Most functions exit in one of two ways:
  - They either return a value, indicating success.
  - Or they throw an error, indicating failure.
- In the next few slides:
  - Return values (implicit versus explicit; visible versus invisible).
  - Errors.
  - Eexit handlers, allowing to run code when a function exits.

Implicit versus explicit returns

- Implicit, where the last evaluated expression is the return value.

```
j01 <- function(x) {
  if (x < 10) {
    0
  } else {
    10
  }
}
j01(5)
#> [1] 0
j01(15)
#> [1] 10
```

- Explicit, by calling `return()`.

```
j02 <- function(x) {
  if (x < 10) {
    return(0)
  } else {
    return(10)
  }
}
```

Invisible values

- Most functions return visibly: calling the function in an interactive context prints the result.

```
j03 <- function() 1
j03()
#> [1] 1
```

- Prevent automatic printing by applying `invisible()` to the last value.

```
j04 <- function() invisible(1)
j04()
```

- Verify that this value exists with `pint` or `()`.

```
print(j04())
#> [1] 1

(j04())
#> [1] 1
```

Invisible values cont'd

- The most common function that returns invisibly is `<-`.

```
a <- 2
(a <- 2)
#> [1] 2
```

- This is what makes it possible to chain assignments.

```
a <- b <- c <- d <- 2
```

- In general, any function called primarily for a side effect (like `<-`, `print()`, or `plot()`) should return an invisible value (typically the value of the first argument).

Errors

- If a function cannot complete its assigned task, it should throw an error with `stop()`:
  - Immediately terminates the execution of the function.
  - Indicates that something has gone wrong, and forces the user to deal with the problem.

```
j05 <- function() {
  stop("I'm an error")
  return(10)
}
j05()
#> Error in j05(): I'm an error
```

- Some languages (like C, Go, and Rust) rely on special return values to indicate problems, but in R you should always throw an error.

Exit handlers


```r
j06 <- function(x) {
  cat("Hello\n")
  on.exit(cat("Goodbye!\n"), add = TRUE)

  if (x) {
    return(10)
  } else {
    stop("Error")
  }
}

j06(TRUE)
#> Hello
#> Goodbye!
#> [1] 10

j06(FALSE)
#> Hello
#> Error in j06(FALSE): Error
#> Goodbye!
```

Exit handlers with `on.exit()`

- Always set `add = TRUE`:
  - If you don't, each call to `on.exit()` overwrites previous ones.
  - Even when only registering a single handler, it's good practice to set `add = TRUE`.
- `on.exit()` is useful because it allows to place clean-up code directly next to the code that requires clean-up.

```
cleanup <- function(dir, code) {
  old_dir <- setwd(dir)
  on.exit(setwd(old_dir), add = TRUE)

  old_opt <- options(stringsAsFactors = FALSE)
  on.exit(options(old_opt), add = TRUE)
}
```

Exit handlers with `on.exit()` cont'd

- Coupled with lazy evaluation, a useful pattern for running a block of code in an altered environment.

```
with_dir <- function(dir, code) {
  old <- setwd(dir)
  on.exit(setwd(old), add = TRUE)

  force(code)
}

getwd()
#> [1] "/home/tvatter/Dropbox/teaching/stat5206/2019_fall/lectures"
with_dir("~", getwd())
#> [1] "/home/tvatter"
getwd()
#> [1] "/home/tvatter/Dropbox/teaching/stat5206/2019_fall/lectures"
```

- `force()` isn't strictly necessary here as simply referring to code will force its evaluation.
- But makes it clear that we are deliberately forcing the execution.