Create your own functions - much simpler!

A function is defined as follows.

```
> quadx <- function(x) {          # function definition, input variables
    x^2 + x + 1                 # R statements, what the function does
  }
```

The return value is the last value computed -- but you can also use the "return" function.

```
> quadx <- function(x) {
  return( x^2 + x + 1 )
}
> fxy <- function(x, y=3) { x+y^2 }
```
Arguments can have default values.

When you call a function and list the argument **names**, you don't have to worry about the order you list them in (this is very useful for functions that expect many arguments -- in particular arguments with default values).

```
> f(y=4, x=3.14)
> f(4, 3.14)
```
without argument names, it will assign x=4, y=3.14

Create your own functions - much simpler!

After the arguments, in the definition of a function, you can put three dots represented the arguments that have not been specified and that can passed through another function (very often, the "plot" function).

```
f <- function(x, ...) {          # ... you could put here col="red", pch=19, or any
  plot(x, ...)                   # other valid argument
}
```

But you can also use this to write functions that take an arbitrary number of arguments:

```
f <- function (...) {
  query <- paste(...)  # Concatenate all the arguments to form a string
  con <- dbConnect(dDriver("SQLite"))
  dbGetQuery(con, query)
  dbDisconnect(con)
}

f <- function (...) {
  l <- list(...)       # Put the arguments in a (named) list
  for (i in seq(along=l)) {
    cat("Argument name:", names(l)[i], "Value:", l[[i]], "\n")
  }
}
```

Functions have NO SIDE EFFECTS: all the modifications are local. In particular, you cannot write a function that modifies a global variable.

Functions

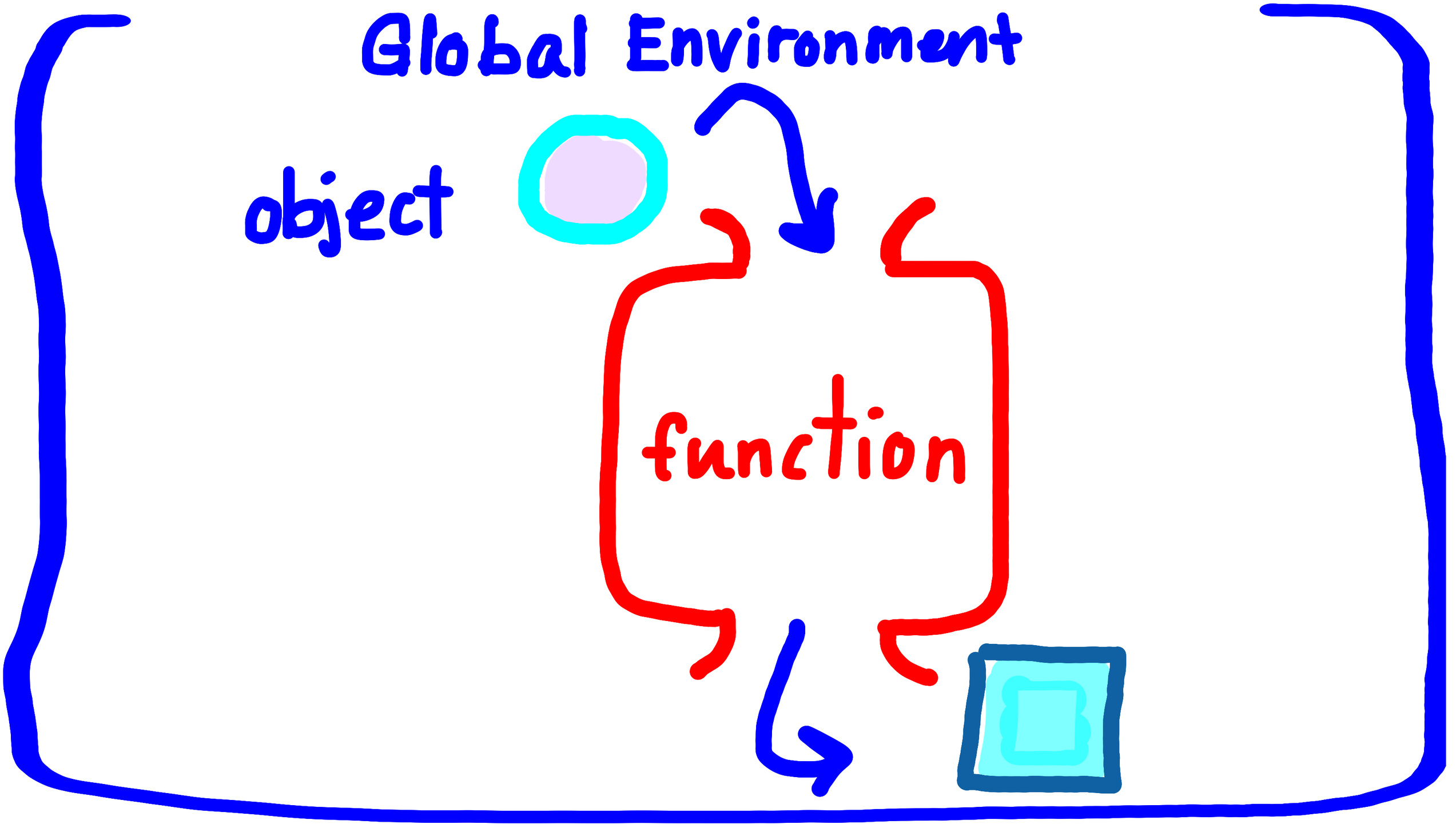
Functions operate in a local scope.

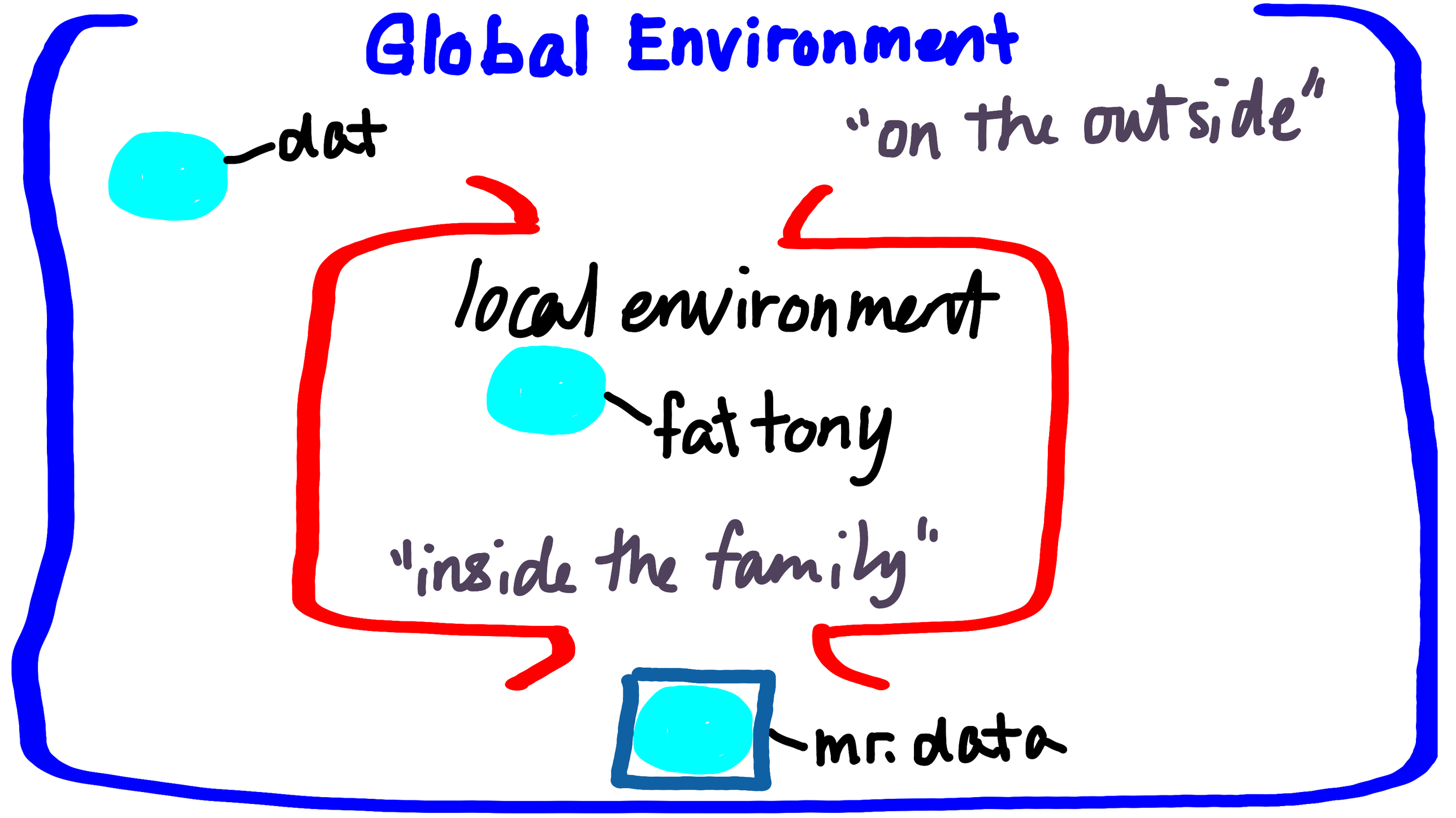
- Globally defined objects are recognized inside functions, but
- Objects defined "inside" of functions are local only.

Simplifies coding - don't have to remember all objects used

Makes functions more portable (won't crash with other functions or global environment).

"Black Box" analogy


Global Environment
object
function


Global Environment
dat
"on the outside"
local environment
fat tony
"inside the family"
mr. data

inside R : functions

To get the code of a function, you can just type its name -- with no brackets.

```
> summary
function (object, ...)        # It is a generic function, that uses different methods
UseMethod("summary")          # depending on the class attribute of the object
<environment: namespace:base>

> methods("summary")         # Here are all the methods that are defined for “summary”

 [1] summary.Date            summary.POSIXct         summary.POSIXlt         summary.aov
 [5] summary.aovlist         summary.connection      summary.data.frame      summary.default
 [9] summary.ecdf*           summary.factor          summary.glm             summary.infl
[13] summary.lm              summary.loess*          summary.manova          summary.matrix
[17] summary.mlm             summary.nls*            summary.packageStatus*  summary.ppr*
[21] summary.prcomp*         summary.princomp*       summary.stepfun         summary.stl*
[25] summary.table           summary.tukeysmooth*

   Non-visible functions are asterisked
```

inside R : functions

Here's how we find out what is inside summary.lm:

```
> summary.lm
function (object, correlation = FALSE, symbolic.cor = FALSE, ...)   {
    z <- object
    p <- z$rank
    if (p == 0) {
        r <- z$residuals
        n <- length(r)
        w <- z$weights
        if (is.null(w)) {
            rss <- sum(r^2)
        }
        else {
            rss <- sum(w * r^2)
            r <- sqrt(w) * r
        }
        resvar <- rss/(n - p)
        ans <- z[c("call", "terms")]
        class(ans) <- "summary.lm"
        ans$aliased <- is.na(coef(object))
        ans$residuals <- r
...
}
```

Note: **Internal** functions are "hidden" inside the namespace of a package -- the programmer has chosen to not make it available to the global environment. To find these, use `getAnywhere("functionname")`

the R environment : Packages

The R programming language is written in **modules** called **packages**, which are groups of related functions organized together in a bundle.

Packages are the means by which R is extended by the open-source community (user contributed packages)

Not all aspects of the R programming language are **loaded** every time you fire R up. -- you have to load optional packages with every new session.

the R environment : Packages

default packages at startup are **base** & a few others.

```
> search()   # gives search path for R objects

 [1] ".GlobalEnv" "tools:RGUI" "package:methods" "package:stats"
 [5] "package:graphics" "package:grDevices" "package:utils"
"package:datasets"
 [9] "Autoloads" "package:base"
```

When you type a command or object name in R, it "searches" through the "search path" for a match and then takes appropriate action (be it the name of a data object, function, operator, etc.).

NOTE: Don't create objects with the same name as R commands! (e.g., t, T, F, c all are special characters) Results are unpredictable

the R environment : Packages

```
> searchpaths()  # gives path to package source code on your
                   # computer's file system

 [1] ".GlobalEnv"
 [2] "tools:RGUI"                                                \
 [3] "/Library/Frameworks/R.framework/Versions/2.4/Resources/library/methods"
 [4] "/Library/Frameworks/R.framework/Versions/2.4/Resources/library/stats"
 [5] "/Library/Frameworks/R.framework/Versions/2.4/Resources/library/graphics"
 [6] "/Library/Frameworks/R.framework/Versions/2.4/Resources/library/grDevices"
 [7] "/Library/Frameworks/R.framework/Versions/2.4/Resources/library/utils"
 [8] "/Library/Frameworks/R.framework/Versions/2.4/Resources/library/datasets"
 [9] "Autoloads"
[10] "/Library/Frameworks/R.framework/Versions/2.4/Resources/library/base"
```

INSTALLING Packages: saves the package source code to the appropriate place in your computer's file directory (usually a place that users don't mess with)

LOADING Packages: attaches the package from your computer's R library to your search path (adds it to your session).

```
> library("packagename")
```

the R environment : attach(), with()

ATTACH: Attach Set of R Objects to Search Path

The database is attached to the R search path. This means that the database is searched by R when evaluating a variable, so objects in the database can be accessed by simply giving their names.

```
> data.frame(xx=1:10,yy=runif(10)) -> ddata
> attach(ddata)
> search()
 [1] ".GlobalEnv"   "ddata"   "tools:RGUI"   "package:methods"
 [5] "package:stats"   "package:graphics"   "package:grDevices"
"package:utils"
 [9] "package:datasets"   "Autoloads"   "package:base"
```

Now ddata is in the search path, so R knows what you mean when you specify "xx" or "yy" without the data.frame.

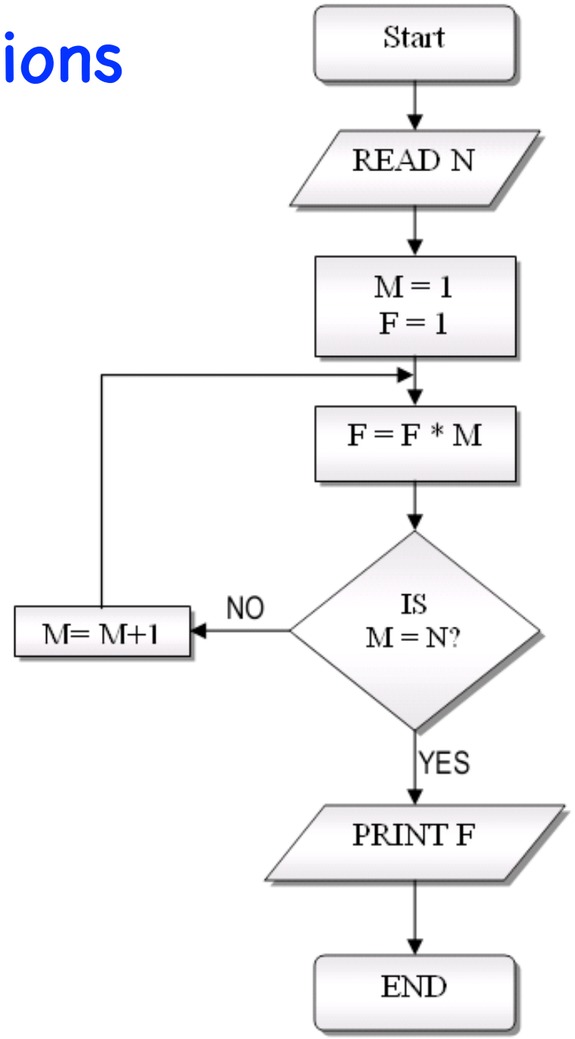
WITH: Creates a tiny local environment for single line of code, using a dataframe. Less confusing that ATTACH

Program Flow

Abstract your programming into distinct actions or **steps**

What has to happen **within** each **step**?

How are the **steps connected**?



Flowchart for computing factorials

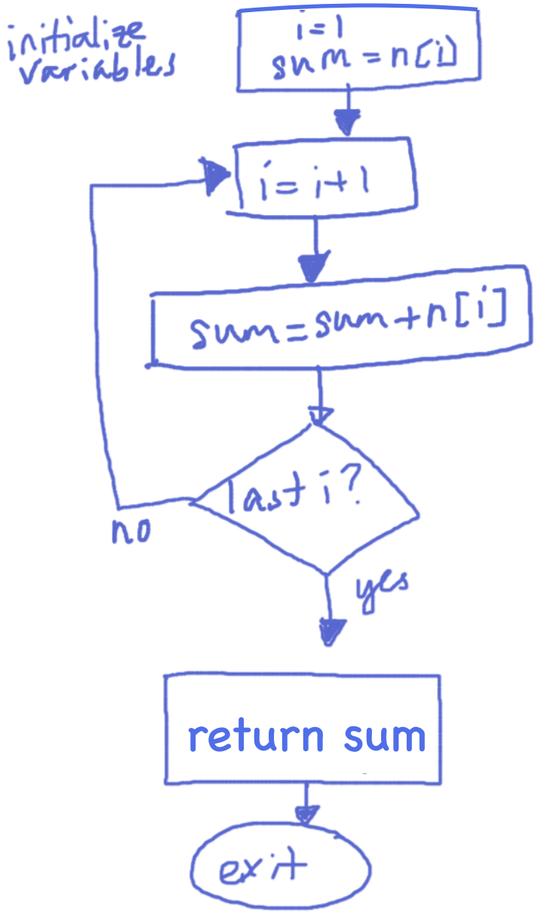
Program Flow

Abstract your programming into distinct actions or **steps**

What has to happen **within** each **step**?

How are the **steps connected**?

Simpler Example

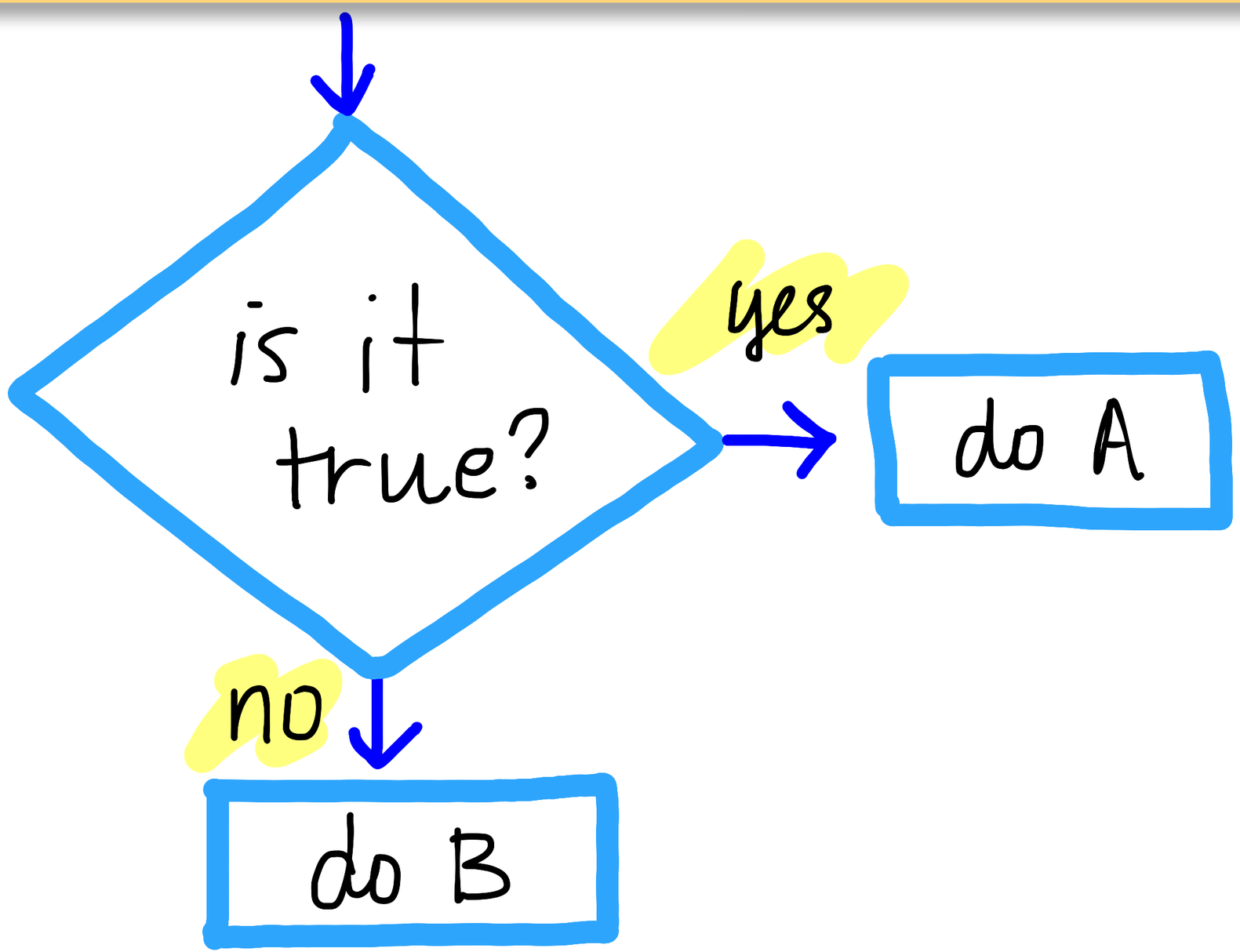


Flowchart for computing sum using a loop

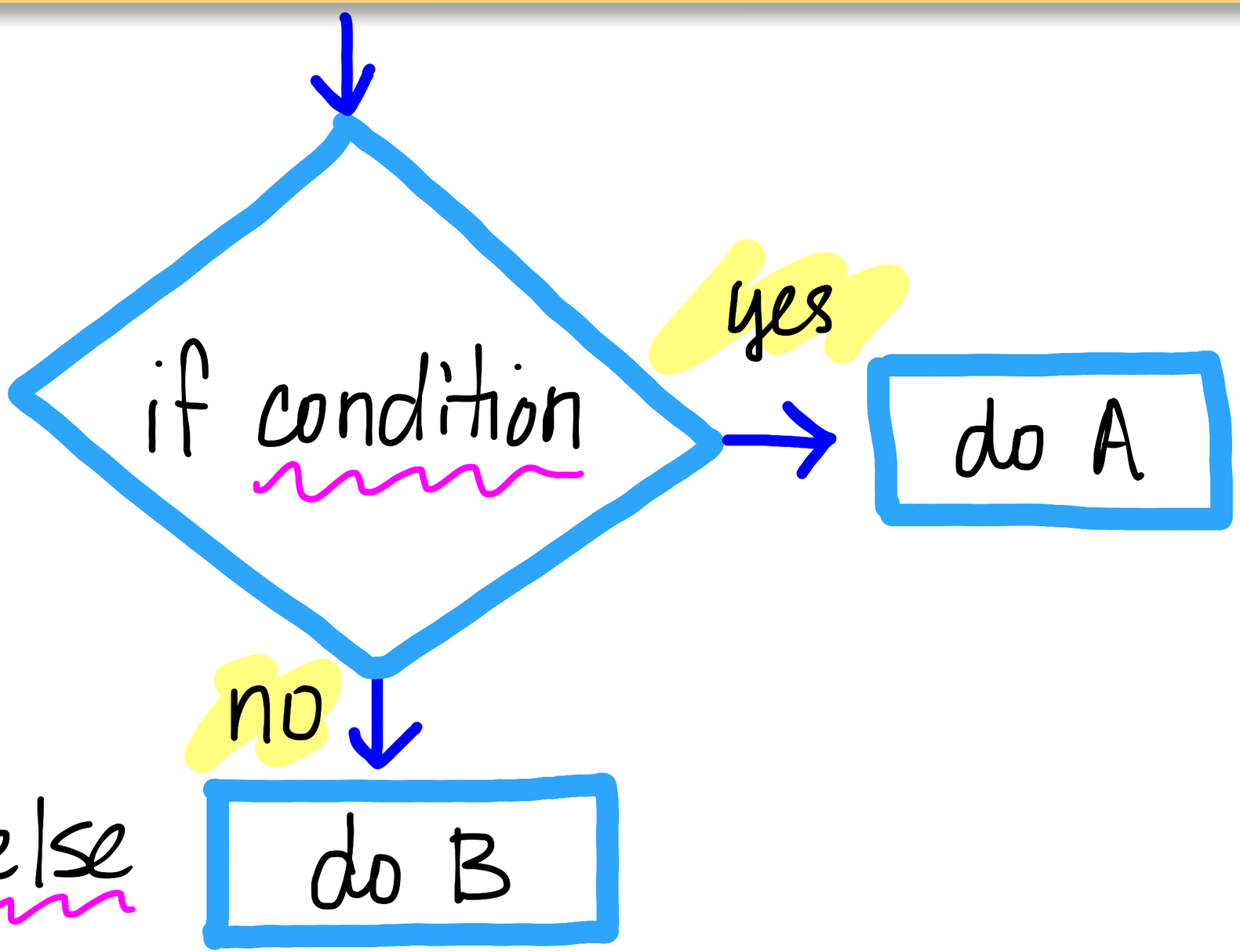
Conditional Statements

`if else`	statement
`while`	loop
`repeat`	loop

Conditional Statements `if else` statement



Conditional Statements `if else` statement



Conditional Statements `if else` statement

if condition statement else statement

```
if   x>y   print(x)
```

else is optional - nothing happens if condition false

```
if   x>y   print(x) else print ("x is too small")
```

if true

else is what happens if condition is false

Conditional Statements

while loop

```
while ( condition ) expression
```

while condition is TRUE allows execution

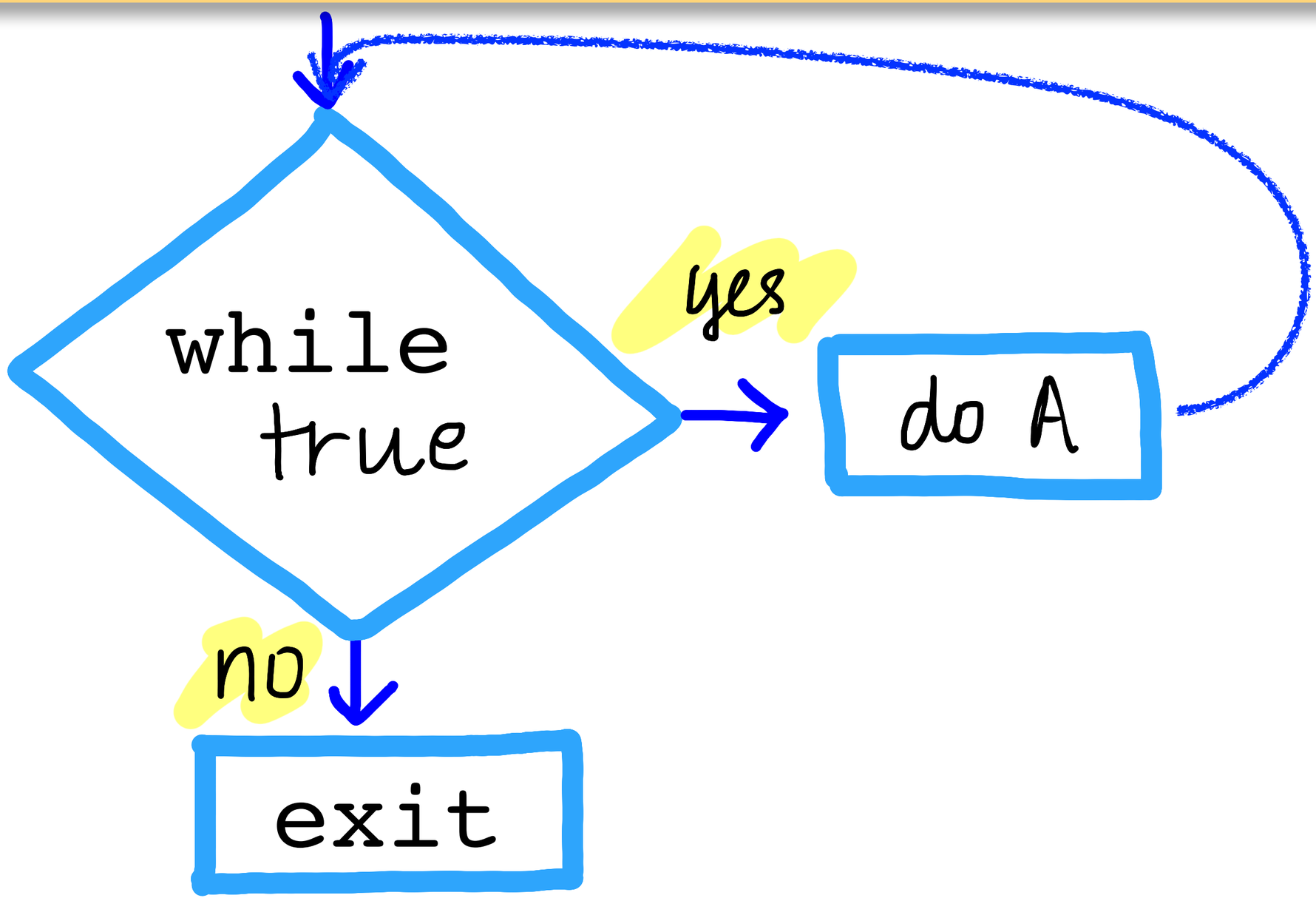
```
while ( tolerance > .001 ) {
  do some more calculations
}
```

tests before executing the expression

so make sure the condition can change (and be FALSE), or you will have an **infinite loop!**

Conditional Statements

while loop



Conditional Statements — repeat loop

```
repeat ( condition ) expression


repeat {

  expressions
  if (condition) { break }

}
```

break is required to stop execution.

Need conditional as well.

Be Careful!!

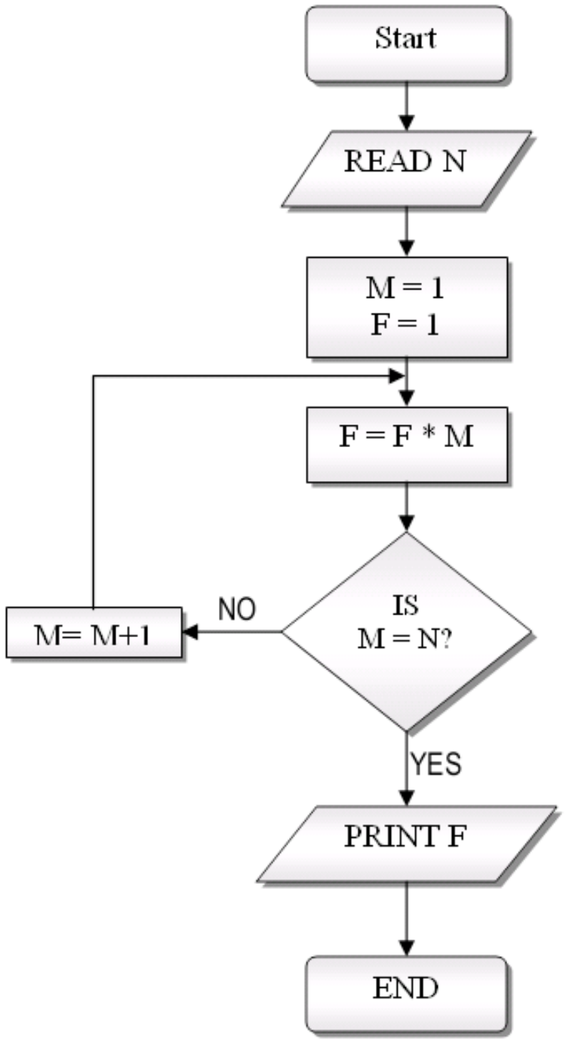
In general, try to avoid while and repeat loops if there is another way.

They can go on forever if you're not careful.

Program Flow

computing a factorial 4*3*2*1

flowchart



pseudo code

target number

initialize objects, start from 1

multiply current with next

are we at target?

yes, fantastic. stop.
no: add 1 and go back to

Vectorized Calculations apply functions

```
x <- matrix( 1:4, nrow=2)
x + 1
```

many simple functions in R are already "vectorized"

Apply allows you to "apply" functions to an entire list, vector, row, or column

```
apply (X, MARGIN, FUN )
```

```
dat <- iris[1:4,]
apply( dat, 2, mean )
```

"2" is the row index, and function is mean, so taking mean over rows

Vectorized Calculations apply functions

Flavors

tapply operates on a "ragged array" (i.e., groups of different sizes, for instance one object indexed by a list of factors)

note: index must be coerced to list

```
tapply (array or vector, index , FUN)
```

```
tapply( iris$Petal.width, list(iris$Species), mean )
                                    # calculate means by species
```

Vectorized Calculations `apply` functions

Flavors

these all operate on components of a list or vector or array

```
apply      #operates over margin (1=rows, 2=columns)
lapply     # returns a list
sapply     # returns "friendly" output, vector or matrix if possible
mapply     # works on multiple  arguments — HEADACHES!
aggregate  # computes summary statistics over subsets of the data
```