PROGRAMMING WITH R

NORATIQAH MOHD ARIFF, UKM

INTRODUCTION

- A computer program is a sequence of commands and instructions to effectively solve a given problem.
- Each computer program is based on an underlying procedure called algorithm.
- An algorithm may be implemented in different ways, leading to different programs using the same procedure.

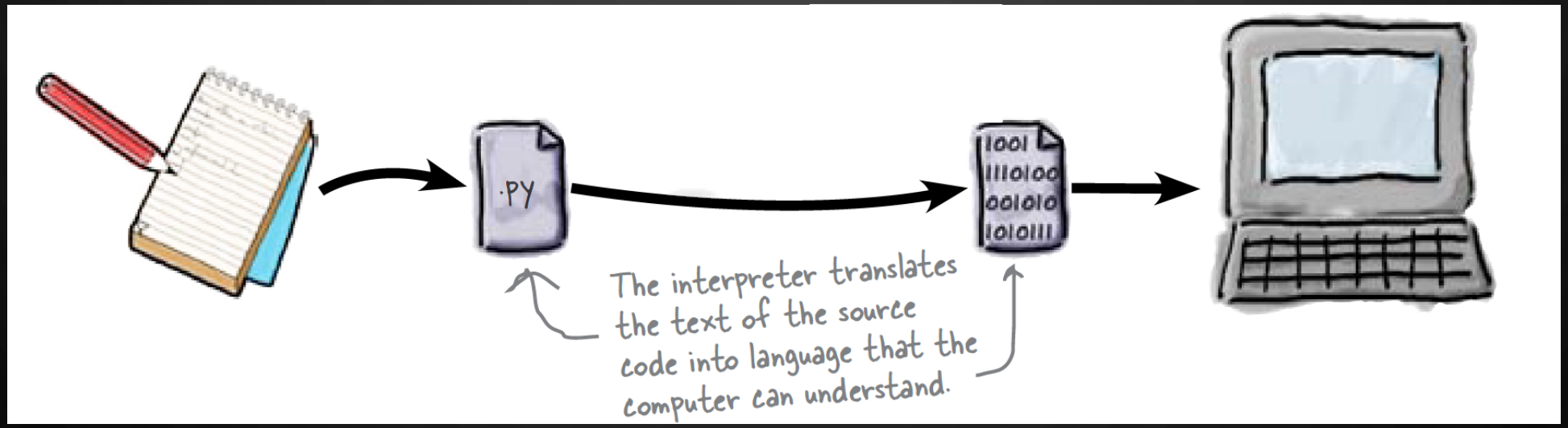
WHY DO PROGRAMMING?

- If you use other people's software, you will always be limited by what other people think you want to do.
- Write your own programs and the only limit will be your own imagination.
- Even with all those programs on your computer, you still need to do something different, something specific to you.

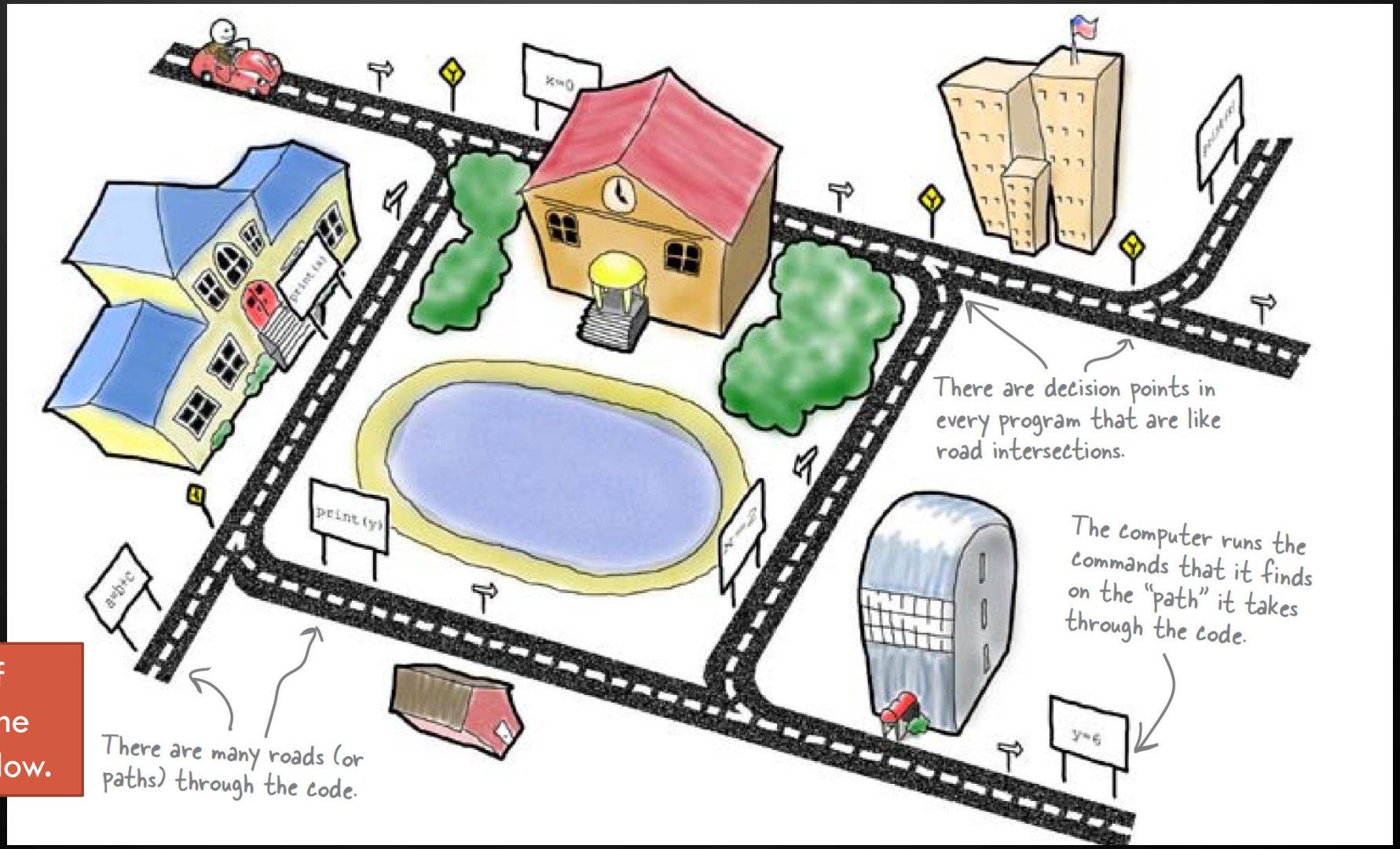
COMMON PROPERTIES

- Programs are usually written by humans and executed by computers. Hence a program should be clear, concise and direct.
- Programs are usually lists of consecutive steps.
- Programs have well-defined initiations and terminations.
- Programs have well-defined constants, inputs, outputs and variables.
- Programs are written by using a finite set of operations defined by programming languages.

HOW DOES PROGRAMMING WORKS?



CODEVILLE



A path is a set of instructions that the computer will follow.

ALGORITHMS AND FLOWCHARTS

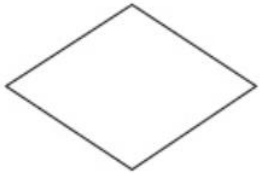
- A typical programming task can be divided into two phases:
- Problem solving phase:
  - Produce an ordered sequence of steps that describe solution of problem
  - This sequence of steps is called an algorithm
- Implementation phase:
  - Implement the program in some programming language

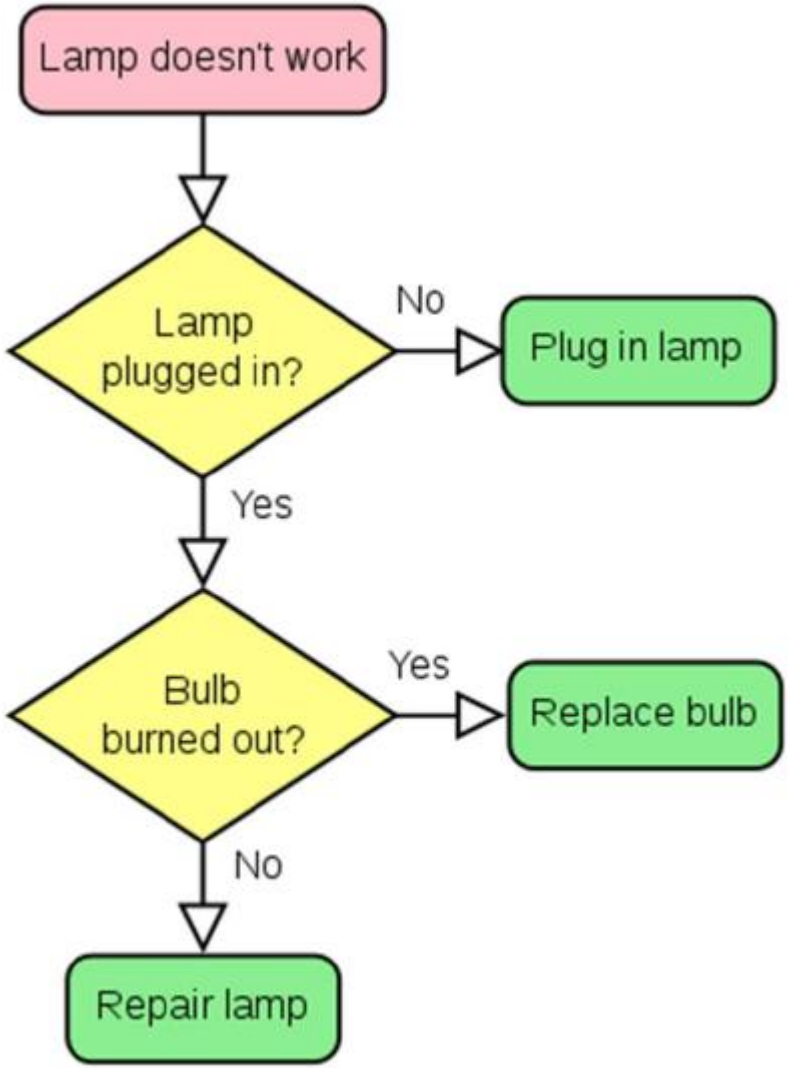
ALGORITHMS AND FLOWCHARTS

- First step in problem solving is to produce a general algorithm (one can use pseudo code).
- Pseudo code is an artificial and informal language that helps programmers develop algorithms.
- Pseudo code is very similar to everyday English.
- Program flowcharts are graphical representations show the sequence of instructions in a single program.
- A flowchart emphasizes individual steps and their interconnections.

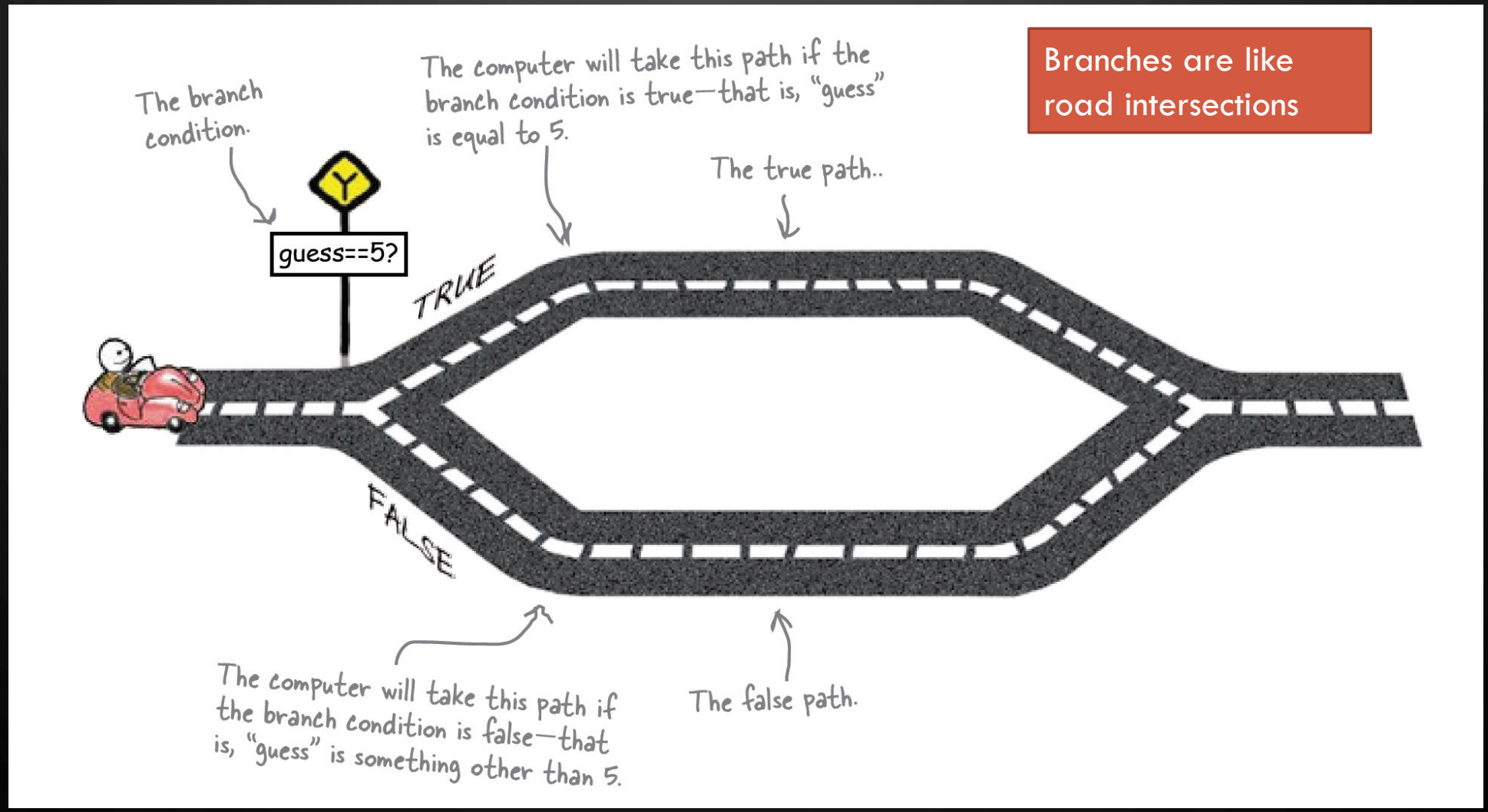
FLOWCHART SYMBOLS

Basic

Name	Symbol	Use in Flowchart
Oval		Denotes the beginning or end of the program
Parallelogram		Denotes an input operation
Rectangle		Denotes a process to be carried out e.g. addition, subtraction, division etc.
Diamond		Denotes a decision (or branch) to be made. The program should continue along one of two routes. (e.g. IF/THEN/ELSE)
Hybrid		Denotes an output operation
Flow line		Denotes the direction of logic flow in the program



IF/ ELSE BRANCHES



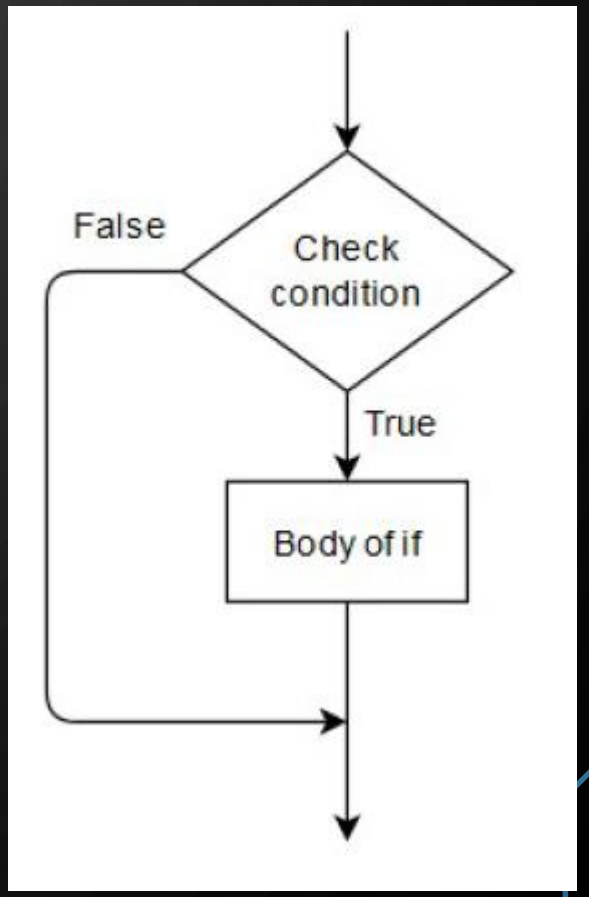
Branches are like road intersections

R IF STATEMENT

```
if (test_expression) {
    statement
}
```

- If the “test_expression” is TRUE, the statement gets executed. But if it is FALSE, nothing happens.
- Example:

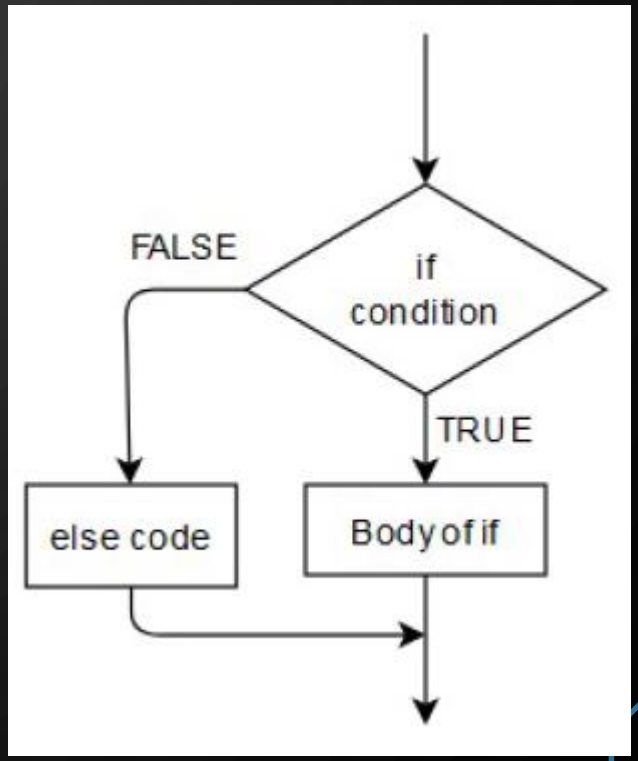
```
x<-5
if(x>0){
    print("x is positive")
}
```



R IF…ELSE STATEMENT

```
if (test_expression) {
    statement1
} else {
    statement2
}
```

- The “else” part is optional and is only evaluated if “test_expression” is FALSE.
- It is important to note that else must be in the same line as the closing braces of the if statement.



R IF…ELSE STATEMENT

- Example:

```
x<-5
if(x>=0){
     print("x is non-negative")
} else{
     print("x is negative")
}
```

R IF…ELSE IF STATEMENT

```
if ( test_expression1) {
    statement1
} else if ( test_expression2) {
    statement2
} else if ( test_expression3) {
    statement3
} else
    statement4
```

- We can use as many if…else if statement as we want as follows.
- Only one statement will get executed depending upon the "test_expressions".

R IFELSE STATEMENT

- There is a vector equivalent form of the if…else statement in R, the ifelse() function.

```
ifelse(test_expression,x,y)
```

- The returned vector has element from “x” if the corresponding value of “test_expression” is TRUE or from y if the corresponding value of “test_expression” is FALSE.

R IFELSE STATEMENT

- Example:

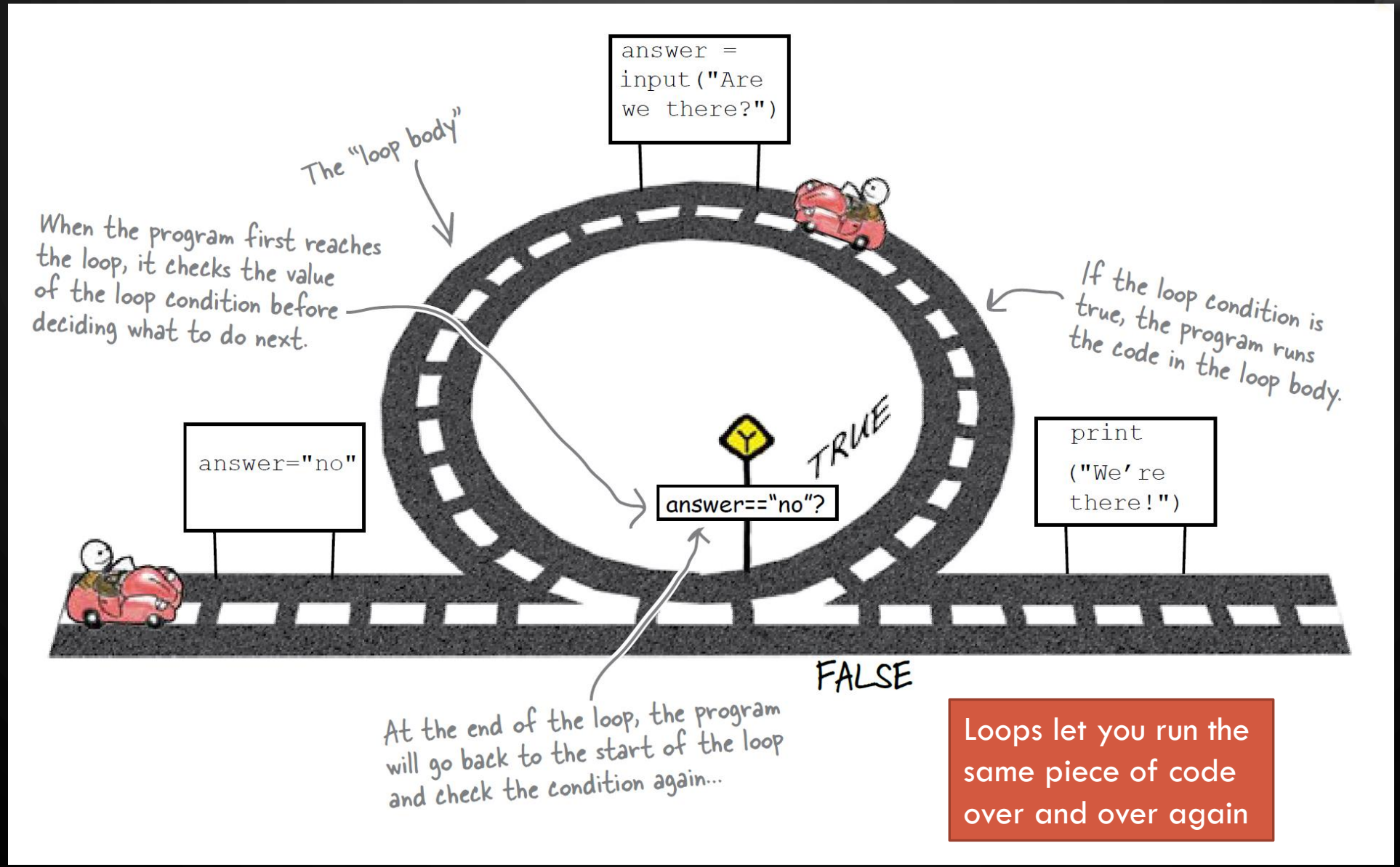
```
a<-c(3,-2,5)
> ifelse(a>=0,"non-negative","negative")
[1] "non-negative" "negative" "non-negative"
```

R SWITCH STATEMENT

- The switch(expr,…) function evaluates “expr” and accordingly chooses one of the further arguments.
- Example:

```
x<-3
switch(x, "red", "blue", "green", "yellow")
[1] "green"
```

LOOP



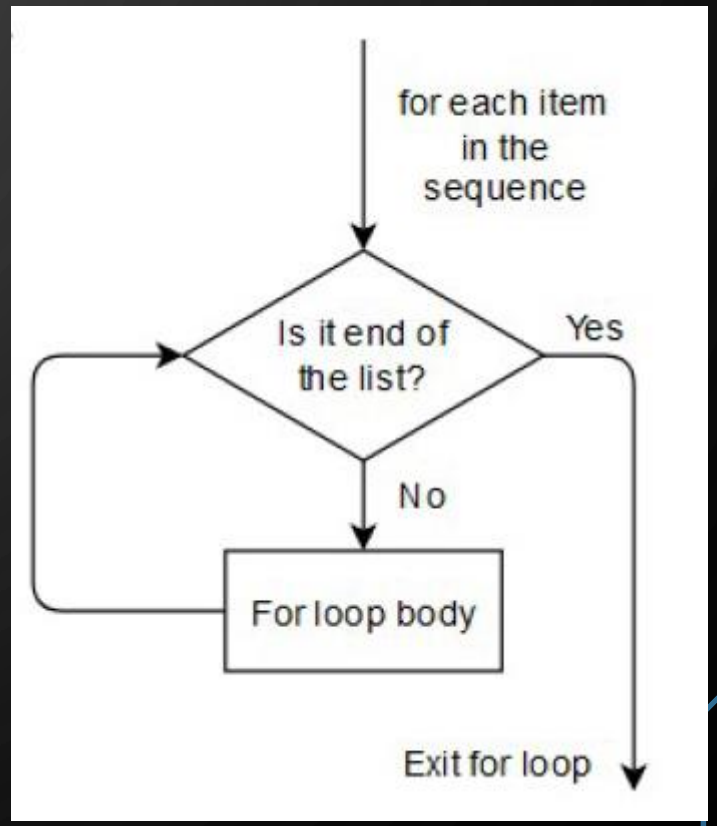
Loops let you run the same piece of code over and over again

R FOR STATEMENT

```
for (val in sequence)
{
     statement
}
```

- A for loop is used to iterate over a vector.
- In each iteration, statement is evaluated.
- Example:

```
x<-1
for(i in 1:3){
x<-x+1
}
x
[1] 4
```

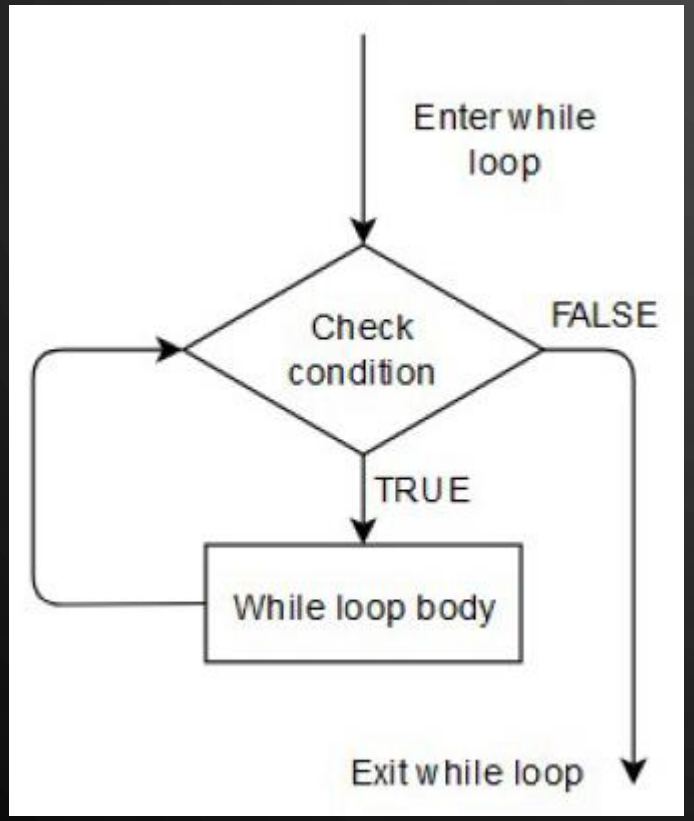


R WHILE STATEMENT

```
while (test_expression)
{
    statement
}
```

- The “test_expression” is evaluated and the body of the loop is entered if the result is TRUE.
- The statements inside the loop are executed and the flow returns to evaluate the “test_expression” again.
- This is repeated each time until “test_expression” evaluates to FALSE, in which case, the loop exits.

R WHILE STATEMENT



- **Example:**

```
x<-4
while(x<=10){
x<-x+1
}
x
 [1] 11
```

R BREAK STATEMENT

- A break statement is used inside a loop to stop the iterations and flow the control outside of the loop.
- Example:

```
x <- 1:5
for (val in x) {
    if (val == 3){
        break
    }
   print(val)
}
[1] 1
[1] 2
```

R NEXT STATEMENT

- A next statement is useful when we want to skip the current iteration of a loop without terminating it.
- Example:

```
x <- 1:5
for (val in x) {
    if (val == 3){
        next
    }
   print(val)
}
```

```
[1] 1
[1] 2
[1] 4
[1] 5
```

PROGRAMMING IN R USING FUNCTIONS

- Programming in R is organized around functions.
- The basic declaration or definition of a function looks like below:

```
myfunction<- function(argument1, argument2, …) {
    # clever manipulations of arguments
    return(values to return)
}
```

input

statements

output

The name of the function

```
myfunction(argument1, argument2, …)
```

Call the function

PROGRAMMING IN R USING FUNCTIONS

- Objects that are created within the function are local to the environment of the function – they don't exist outside of the function.
- You can pass the values into the global environment with the return() function.
- The argument can be any type of object (scalar, matrix, vector, data frame or logical)
- A function needs to have a name and a body of code that does something.

EXAMPLE

```
square.it<-function(x){
   square<-x*x
   return(square)
}
#square a number
square.it(5)
#square a vector
square.it(c(1,4,2))
```

EXAMPLE

```
my.fun<-function(X.matrix, y.vec, z.scalar){
#use previous function
sq.scalar<-square.it(z.scalar)

mult<-X.matrix%*%y.vec
Final<-mult*sq.scalar
return(Final)
}
my.mat<-cbind(c(1,2,3),c(3,4,5))
my.vec<-c(5,6)

my.fun(X.matrix=my.mat, y.vec=my.vec, z.scalar=9)
```

GOOD FUNCTION WRITING PRACTICES

- Keep your functions short.
  - If things start to get long, you can probably split up the function into several functions.
  - It also makes your code easier to update.
- Put in comments on what are the inputs, what the function does and what is the output.
- Check for errors along the way.

THANK YOU