Flow Control

301113 Programming for Data Science

WESTERN SYDNEY
UNIVERSITY

School of Computer, Data and Mathematical Sciences

Week 3

Outline

Motivation

Your team will be provided with data containing the number of requests sent to a set of servers over a period of time. You have been asked to write a script to process this data. You script needs to:

1. Read in the data.
2. For each item, test if it is an outlier.
3. Report which items are outliers.

Note that the number of data items is not known until the data arrives.

Outline

Top-down Design

To convert our problem into an R script, we use top-down design

1. Define the input and output of the program.
2. Design the algorithm.
3. Convert the algorithm into R code.
4. Test the program.

For our problem, the input is a set of numbers (representing the number of requests to servers). The set can be any size, but we will assume that it is at least size 1.

The output is the set of ids representing the position of the outliers in the input.

Pseudocode

Pseudocode is a plain language description of an algorithm. It is not intended to be run by a computer. The purpose is to allow us to layout the design of the algorithm.

Pseudocode

Pseudocode is a plain language description of an algorithm. It is not intended to be run by a computer. The purpose is to allow us to layout the design of the algorithm.

```
serverRequests <- input # a vector of request counts

# compute the threshold based on all values in serverRequests
threshold <- compute the threshold using the serverRequests
outlierIds <- start with an emptyVector # create the empty vector to store outlier IDs

for each requestCount in serverReqests {

    if requestCount > threshold {
        ## add the position of the count to the outlier set
        outlierIds <- (outlierId, requestCountId)
    }
}

return(outlierIds)
```

Missing Constructs

To implement this algorithm, we need to be able to iterate though each item in the vector (for loop) and only run parts of the code if a condition is true (if branch).

We will investigate these constructs.

Outline

Making Decisions

So far, we have written sequential programs (each line of the script is run from top to bottom).

The power of computer programs comes from the way which selective pieces of code can be run or repeatedly run, dependent on the input to the program.

We will be examining the flow control methods of branching and loops to obtain more complex algorithms.

The decision to branch or loop is based on the truth of certain statements, so we will first example the logical type and logic statements.

Logical Type

We have seen numeric and string variables. We now introduce logical variables.

Logical variables can contain on the values `TRUE` and `FALSE` (keywords used to represent true and false outcomes). Note that the values `TRUE` and `FALSE` are not numbers or strings (they don't need quotes).

```
happy <- TRUE
class(happy)
```

```
[1] "logical"
```

Logical Type

We have seen numeric and string variables. We now introduce logical variables.

Logical variables can contain on the values `TRUE` and `FALSE` (keywords used to represent true and false outcomes). Note that the values `TRUE` and `FALSE` are not numbers or strings (they don't need quotes).

```
happy <- TRUE
class(happy)
```

```
[1] "logical"
```

Mixing Logical and Numeric

If logical and numeric vectors are mixed (e.g. multiplied or placed in the same vector), then the logical vectors are coerced to numeric, where `TRUE` becomes 1 and `FALSE` becomes 0.

Relational Operators

Relational operators are binary operators which compare the values and provide a logical result.

- `x < y`: x less than y
- `x > y`: x greater than y
- `x <= y`: x less than or equal to y
- `x >= y`: x greater than or equal to y
- `x == y`: x equal to y
- `x != y`: x not equal to y

Comparing strings

Strings can be compared. The order is dependent on the character set used by the machine (e.g. ASCII).

Example Relational Comparisons

Numeric values can be compared:

```
small = 3
large = 10
small <= large
```

```
[1] TRUE
```

```
small != large
```

```
[1] TRUE
```

Strings can also be compared (but the order depends on the machines character set):

```
small = "beef"
large = "cheese"
small < large
```

```
[1] TRUE
```

Even Numbers

Problem

Write a statement that returns `TRUE` if `x` is even.

Problem with Finite Precision of Numeric Variables

Recall that variables are stored in the main memory, which has limited size, and so variables have limited precision. Real numbers are actually represented in kind of scientific notation. Therefore only a limited number of decimal places are stored.

We can demonstrate this using π. The true value of π has an infinite number of decimal places, but R can only store a finite amount in memory.

```
sqrt(pi)^2 == pi # these should be equal
```

```
[1] FALSE
```

If we want to test if two numeric values are equal, they might not be equal due to the finite precision provided.

Testing for Numeric Equality

We showed that two numeric variables that should be equal, might not be equal due to the finite precision offered to variables.

Rather than testing if two numeric variables are equal, we should instead test to see if they are very close to each other.

```
x = sqrt(pi)^2
abs(x - pi) < 1e-8 # test if their difference is very small
```

```
[1] TRUE
```

This allows for the variation in the values due to their finite representation.

Logical Operators

Logical operators compare logical variables and return a logical variable. These are also know as Boolean Operators.

- `!x`: not x. True if x is false.
- `x & y`: $x \cap y$ (x AND y). True only if x and y are true.
- `x && y`: $x \cap y$ (x AND y) short-circuit. True only if x and y are true.
- `x | y`: $x \cup y$ (x OR y). True if either x or y are true.
- `x || y`: $x \cup y$ (x OR y) short-circuit. True if either x or y are true.
- `xor(x, y)`: $x \oplus y$ (x XOR y). True only if either x or y are true, but not both.
- `isTRUE(x)`: returns `TRUE` if x is true.
- `isFALSE(x)`: returns `TRUE` if x is false.

Example Logical Comparisons

Numeric values can be compared:

```
small = 3
large = 10
!(small > large)
```

```
[1] TRUE
```

```
(small < large) & (small > large)
```

```
[1] FALSE
```

```
(small < large) | (small > large)
```

```
[1] TRUE
```

Even Numbers or Negative

Problem

Write a statement that returns `TRUE` if `x` is even or negative.

Short-Circuit Logic Operators

The short-circuit operators will only examine the second argument if required. This is useful if the second argument requires computation to test for its truth (not evaluating it will save time).

```
## If x is divisible by 4, then it must be divisible by 2,
## so there is not point evaluating the second term.
(x %% 4 == 0) && (x %% 2 == 0)
```

Many programs use short-circuit logic to check that a first function has run successfully before running a second function.

Outline

Selecting Blocks using Logic

Part of the pseudocode we wrote had a branching construct:

```
if requestCount > threshold {
    ## add the position of the count to the outlier set
    outlierIds <- (outlierId, requestCountId)
}
```

We read this as "run the code `outlierIds <- (outlierId, requestCountId)` only if `requestCount > threshold`."

Selecting Blocks using Logic

Part of the pseudocode we wrote had a branching construct:

```
if requestCount > threshold {
    ## add the position of the count to the outlier set
    outlierIds <- (outlierId, requestCountId)
}
```

We read this as "run the code `outlierIds <- (outlierId, requestCountId)` only if `requestCount > threshold`."

Using logic statements, we are able to write an algorithm that runs blocks of code based on the truth of the statement.

We will examine the branching constructs:

- `if` construct
- `switch` construct

if Construct

We can conditionally run blocks of code using the `if` construct. It has the form:

```
if (condition) {
    ## code to run if condition is true
    ## code to run if condition is true
}
```

Example

```
x = 10
if ((x %% 2) == 0) {
    print("x is even")
}

[1] "x is even"
```

Style

The code within the `if` statement was indented, this is not needed, but it helps to make the code more readable.

if and else

The `if` construct also contains an optional `else` terms that allows code to run when the condition is not met.

```
if (condition) {
    ## code to run if condition is true
    ## code to run if condition is true
} else {
    ## code to run if condition is true
}
```

Example

```
x = 9
if ((x %% 2) == 0) {
    print("x is even")
} else {
    print("x is odd")
}

[1] "x is odd"
```

Nesting `if` statements

If statements can be nested to provide a binary decision tree.

Example

```
if (animal == "dog") {
    if (age < 2) {
        status <- "puppy"
    } else {
        status <- "dog"
    }
} else if (animal == "cat") {
    if (age < 2) {
        status <- "kitten"
    } else {
        status <- "cat"
    }
}
```

Style

Code becomes hard to read when nested too deep. Try not to nest code deeper than three levels.

Rectified Linear Unit

Problem

A rectified linear unit (ReLU) is used as an activation function in neural networks. It has the form:

$$z = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Write the R code to compute z when given x.

switch Construct

A switch is a constrained form of an if statement. It is able to run a selected block of code based on the content of a variable. It has the form:

```
z <- switch(x,
    case1 = { # run if x matches case1 },
    case2 = { # run if x matches case2 },
    ...
    { run if no cases matched }
    )
```

Example

```
z <- switch(x,
    dog = "puppy",
    cat = "kitten",
    pig = "piglet"
)
```

Choice of branch

A switch can be replaced with an if statement, and so switches are used less frequently.

Problem

Write the previous switch example as an `if` statement.

Outline

Iterating over blocks of code

There may be times where we want to run a block of code many times, and also allow for variables to change, so that the result of the code execution changes.

In our example, we want to check if each of the items in our vector is an outlier. We could write the code to check each item in the vector (repeating the same code many times), but this will lead to later errors, since we don't know how long the vector of counts will be.

We introduce two methods of looping code:

- `for` construct
- `while` construct

for Construct

The `for` construct requires a vector and a dummy variable name that represents an element of the vector for each iteration of the loop. The code runs for each element in the vector.

```
for (item in vector) {
    ## do something, usually with the item
}
```

The for loop:

1. assigns the first element of `vector` to the variable `item`, then runs the code within the braces.
2. assigns the second element of `vector` to the variable `item`, then runs the code within the braces.
3. assigns the third element of `vector` to the variable `item`, then runs the code within the braces.

and so on, until all items in `vector` have been used.

Example for loop

Sum the integers from 1 to 10.

```
x <- 0
for (a in 1:10) {
    x <- x + a
}
print(x)
```

```
[1] 55
```

Print text.

```
colours <- c("red", "green", "blue", "yellow")
for (colour in colours) {
    cat("I like the colour", colour, "\n")
}
```

```
I like the colour red
I like the colour green
I like the colour blue
I like the colour yellow
```

ReLU again

Problem

We previously wrote the code to compute the output of a rectified linear unit (ReLU).

$$z = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases}$$

Write a for loop that computes the output of a ReLU for the values -5 to 5 (intervals of size 1).

`while` Construct

The `while` construct loops over code, but rather than iterating over a vector, the loop continues while the condition is true.

```
while (condition) {
    ## keep running this code while the condition is true.
}
```

The while loop:

1. checks the condition, if it is true the block of code is run.
2. checks the condition, if it is true the block of code is run.
3. checks the condition, if it is true the block of code is run.

and so on, until the condition is false.

Infinite Loop

If the variables in the condition do not change after one iteration, then the loop will iterate until the program is killed (use Ctrl-c).

Example while loops

Sum the numbers from 1 to 10

```
x <- 0
y <-0
while (x <= 10) {
    y <- x + y
    x <- x + 1 # x is acting as a counter
}
print(y)
```

```
[1] 55
```

Run until the change is zero

```
previous_x <- 100
x <- 10
while(abs(x - previous_x) > 1e-6) {
    previous_x <- x # update previous x
    x <- x - 0.5*x
}
```

break and next statements

The break and next statements can be used in both for and while loops.

- break exits the loop.
- next leaves the current iteration and starts the next.

Example

Sum the elements of vector w.

```
x <- 0
for (a in w) {
    if (is.na(a)) {  # new function!
        print("NA found!")
        break
    }
    x <- x + a
}
print(x)
```

break and next statements

The break and next statements can be used in both for and while loops.

- break exits the loop.
- next leaves the current iteration and starts the next.

Example

Sum the elements of vector w.

```
x <- 0
for (a in w) {
    if (is.na(a)) {  # new function!
        print("NA found!")
        break
    }
    x <- x + a
}
print(x)
```

What's wrong with the above code?

Ignoring the NAs

Problem

The previous example showed how to sum a vector, but print a message if there is an NA. Write the code to sum the vector, but ignore NAs. E.g. the vector sum of (1, 2, NA, 3) is 6.

Bringing it all together

Example

Using what we have learnt, let's write a script to:

1. Read in any number of values.
2. Compute an outlier threshold from the sample.
3. Test if each item in the sample is greater than the threshold.
4. Return a vector containing the ids of the outlier values.

For this task, we define an outlier as a point that is more than 1.5 interquartile ranges past the third quartile.
Recall that :

- the interquartile range can be computed using the function `IQR`.
- the third quartile can be computed using `quantile(vector, 0.75)`

Test the script using the server request count `c(89, 67, 78, 93, 102, 78, 204, 86)`

Summary

- Top-down design helps us to think about our algorithm using abstract blocks.
- Pseudocode is help us to write our algorithm as code.
- Branching constructs allow us to selectively run code based on the contents of variables.
- Loop constructs allow us to repeat blocks of code based on the contents of variables.
- Logical statements provide us with a methods to test conditions for branching and looping.