Programming for Data Analytics

1. Introduction to R and Atomic Vectors

Dr. Jim Duggan,
School of Engineering & Informatics
National University of Ireland Galway.

https://github.com/JimDuggan/CT5102

Instructor – Dr. Jim Duggan

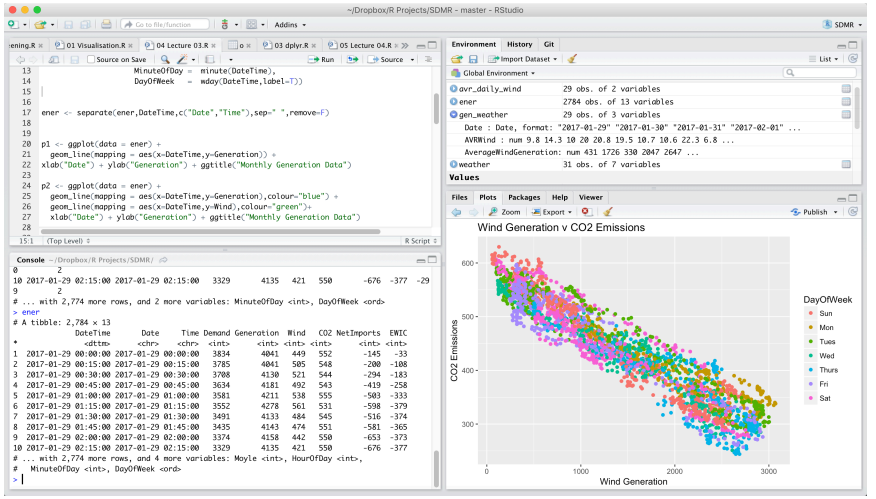
- Senior Lecturer @ NUI Galway
- Research interests:
  - System Dynamics
  - Computational Epidemiology
  - Data Science & Artificial Intelligence
- Industry Experience as Programmer, Software Design, Business Analytics
- Application areas: Public Health Systems

https://twitter.com/_jimduggan

Course Structure

- Weekly Lectures
  - Tuesday 9-11
  - IT101 (Lab-based lectures)
- Lecture Preparation
  - Slides
  - Blackboard (Short Videos)
- Laboratory Work
  - Thursdays 2-4
  - Finnegan Suite

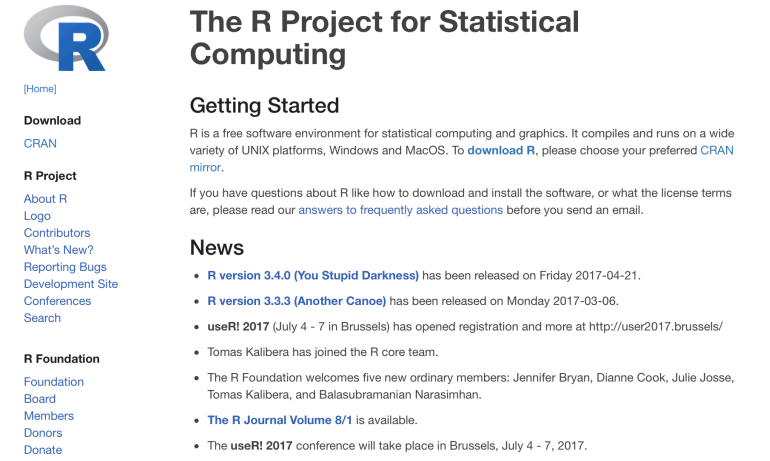
- 10 Assignments x 10 marks each (40% of grade)
- Due Lab Day + 10 , 23.59



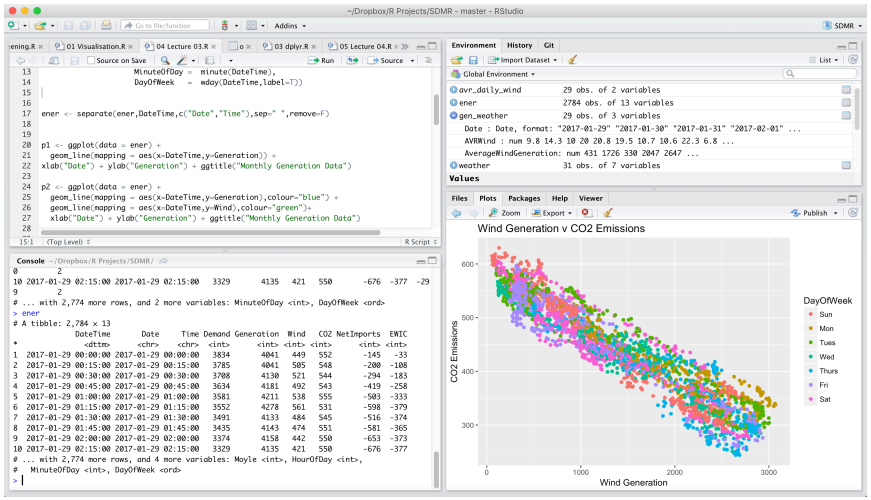
R

- R's *mission* is to enable the best and most thorough exploration of data possible (Chambers 2008).
- It is a dialect of the S language, developed at Bell Laboratories
- ACM noted that *S "will forever alter the way people analyze, visualize, and manipulate data"*

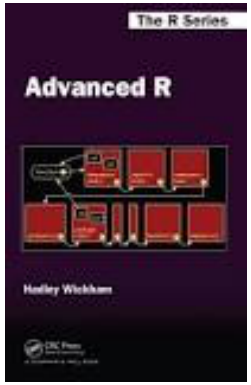
https://www.r-project.org



https://www.rstudio.com



Course Structure

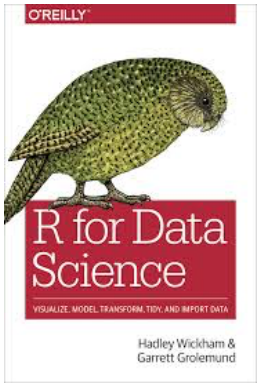


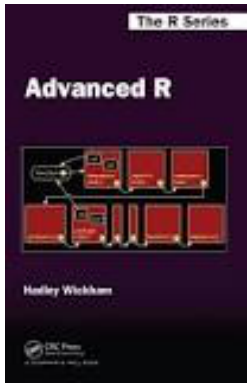
Advanced R

Closures – S3 – S4 – RC Classes – R Packages – RShiny

Data Science

ggplot2 – dplyr – tidyr – stringr – lubridate – Case Studies

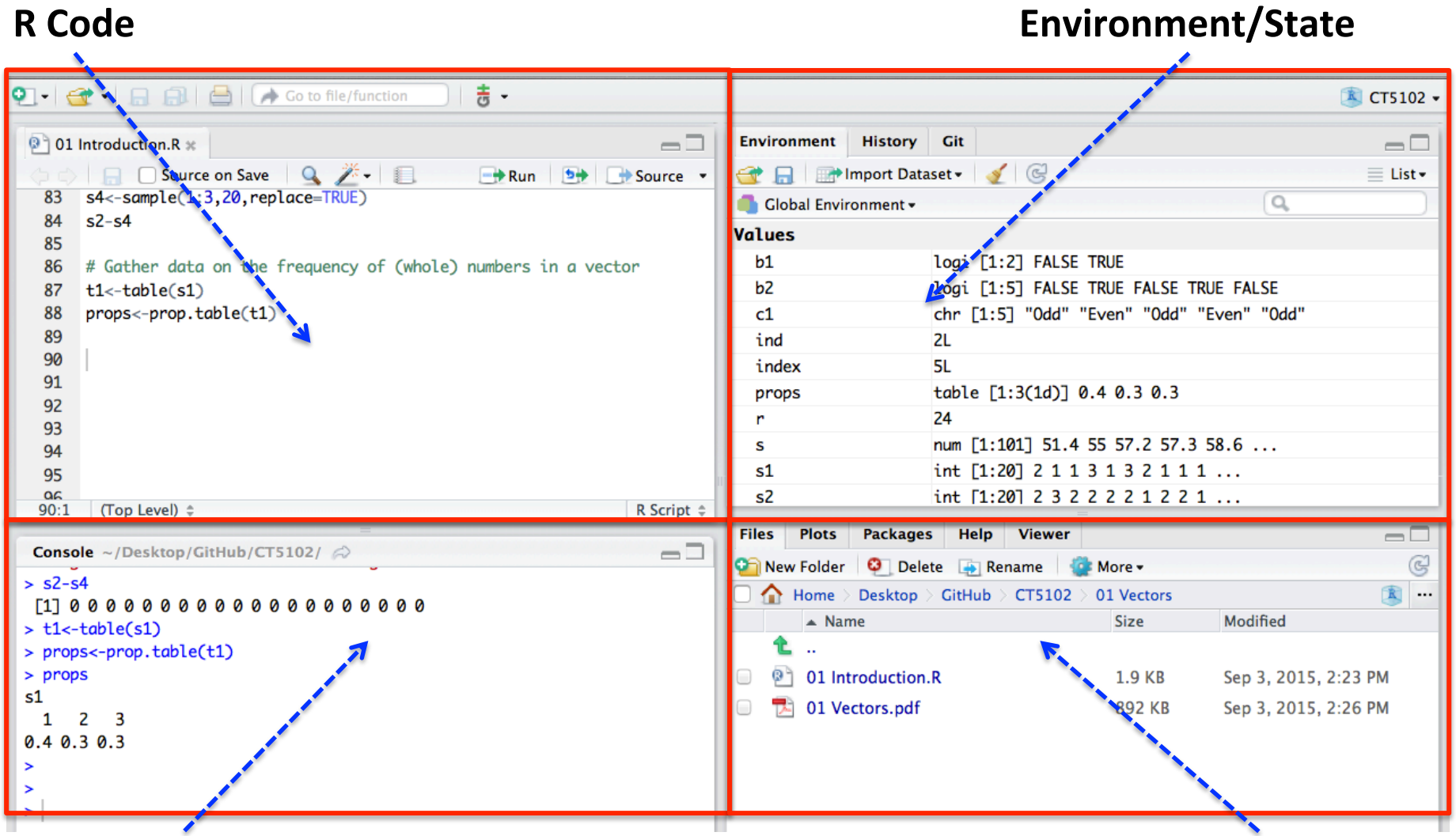




Base R

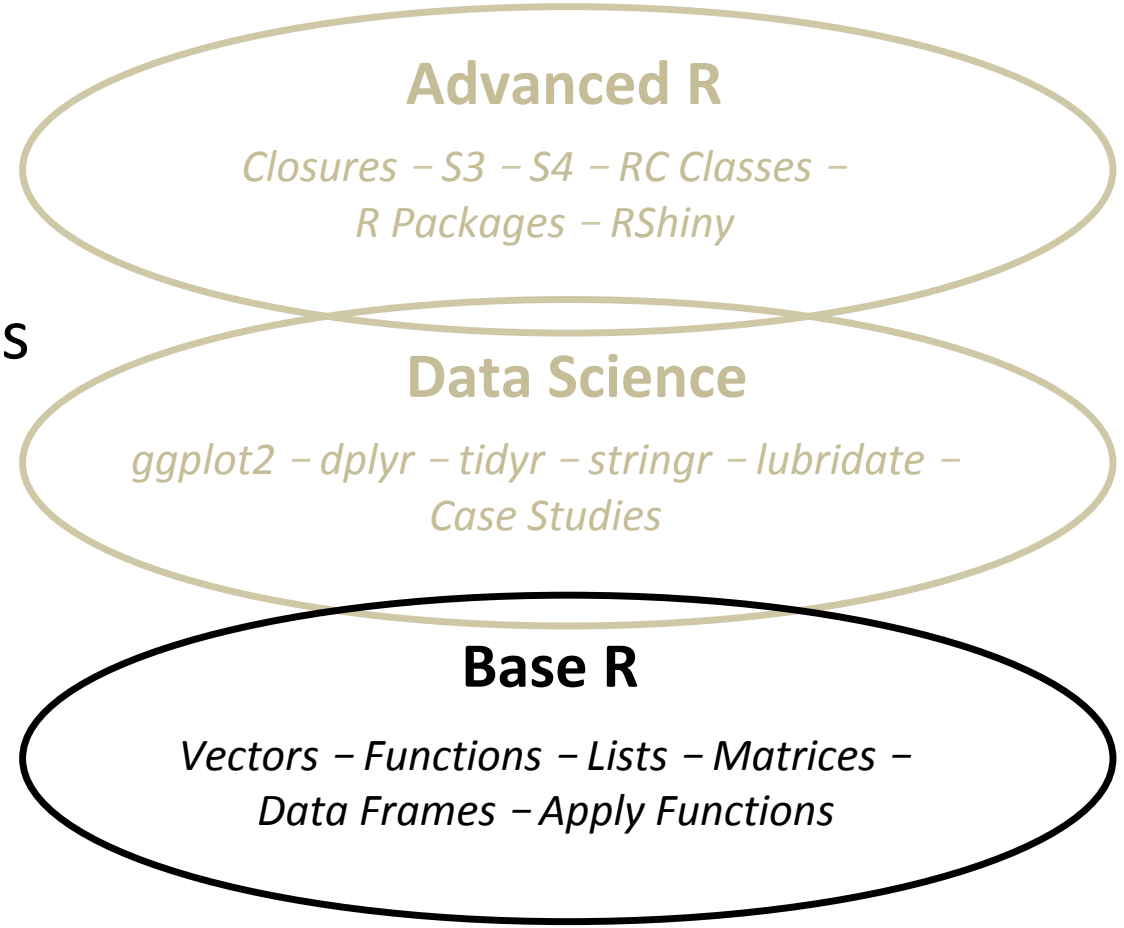
Vectors – Functions – Lists – Matrices – Data Frames – Apply Functions

IDE Used: R Studio



Lecture 01 - Overview

1. R Data Types
2. Atomic Vectors
3. Subsetting Vectors
4. Vectorisation
5. Programming Constructs in R



(1) R – Data Types

	Homogenous	Heterogenous
1d	**Atomic Vector**	List
2d	Matrix	Data Frame
nd	Array	

- The basic data structure in R is the Vector
- Vectors come in two flavours
  - Atomic vectors
  - Lists
- With atomic vectors, all elements have the same type

(2) Atomic Vectors

- Four common types
  - logical
  - integer
  - double (or numeric)
  - character
- Usually created with **c() – short for combine**

```
> dbl_var <- c(2.2, 2.5, 2.9)
> str(dbl_var)
 num [1:3] 2.2 2.5 2.9
>
> int_var <- c(0L, 1L, 2L)
> str(int_var)
 int [1:3] 0 1 2
>
> log_var<- c(TRUE, TRUE, F, FALSE)
> str(log_var)
 logi [1:4] TRUE TRUE FALSE FALSE
>
> chr_var<- c("CT5102","CT561")
> str(chr_var)
 chr [1:2] "CT5102" "CT561"
```

Atomic vector types

```
> dbl_var
[1] 2.2 2.5 2.9
> typeof(dbl_var)
[1] "double"
>
> int_var
[1] 0 1 2
> typeof(int_var)
[1] "integer"
```

```
> log_var
[1]  TRUE  TRUE FALSE FALSE
> typeof(log_var)
[1] "logical"
>
> chr_var
[1] "CT5102" "CT561"
> typeof(chr_var)
[1] "character"
```

Creating atomic vectors using sequences

The colon operator (:) generates regular sequences (atomic vectors) within a specified range.

```
> v1<-1:10
> v1
 [1]  1  2  3  4  5  6  7  8  9 10

> v2<-3:13
> v2
 [1]  3  4  5  6  7  8  9 10 11 12 13
```

Creating vectors of fixed size

- Useful when using loops to generate results

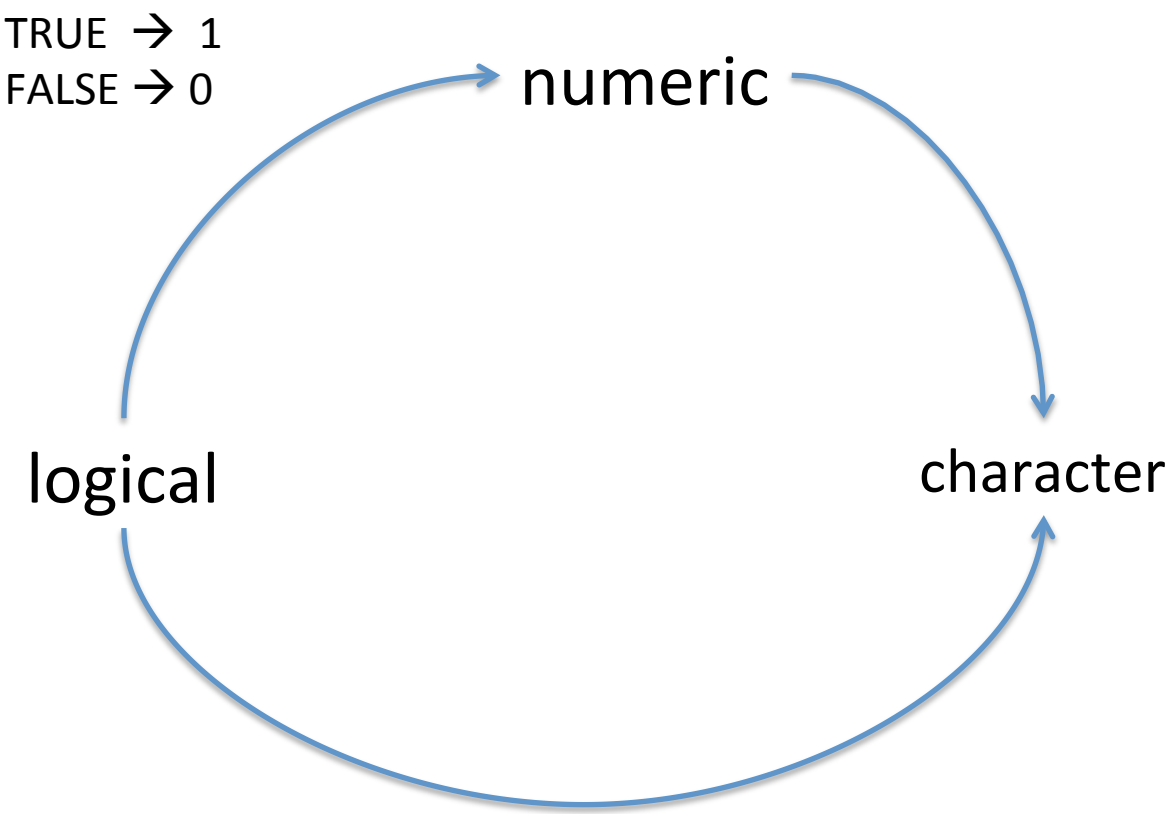
```
> v1 <- vector (mode="numeric", length=20)
>
> v1
 [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
>
> v2 <- vector (mode="character", length=20)
>
> v2
 [1] "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" "" ""
>
> v3 <- vector (mode="logical", length=20)
>
> v3
 [1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
```

Coercion of atomic vectors

- All elements of an atomic vector MUST be of the same type
- When different type are combined, **they will be coerced into the most flexible types**
- What will be the type and values of
  - c(1L, T, F)
  - c(1,T,F)

Coercion Rules
Least to most flexible

- logical
- numeric
- character



Grolemund (2014) p 52

Challenge 1.1

- Determine the types for each of the following (coerced) vectors

```
v1<- c(1L, T, FALSE)

v2<- c(1L, T, FALSE, 2)

v3<- c(T, FALSE, 2, "FALSE")

v4<- c(2L, "FALSE")

v5<- c(0L, 1L, 2.11)
```

(3) Subsetting Atomic Vectors

- Subsetting data is a key activity in data science
- R's subsetting operators are powerful and fast
- For atomic vectors, the operator [is used
- In R, the index for a vector starts at 1

```
> x<-c(2.1, 4.2, 3.3, 5.4)
>
> x
[1] 2.1 4.2 3.3 5.4
>
>
> x[1]
[1] 2.1
> x[c(1,3)]
[1] 2.1 3.3
```

(1) Positive integers

Positive integers return elements at the specified position


```
> x<-1:5
>
> x
[1] 1 2 3 4 5
>
> x[1:2]
[1] 1 2
>
> x[5]
[1] 5
>
> x[5:1]
[1] 5 4 3 2 1
```

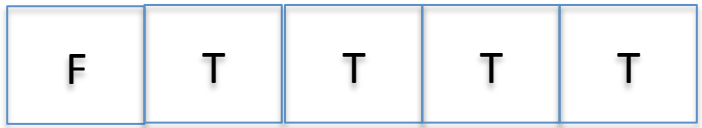
(2) Negative integers

Negative integers omit elements at specified positions

```
> x
[1] 1 2 3 4 5
>
> x[-1]
[1] 2 3 4 5
>
> x[-(3:4)]
[1] 1 2 5
```

(3) Logical Vectors

Select elements where the corresponding logical value is TRUE. This approach supports ***recycling***




```
> x
[1] 1 2 3 4 5
> x[c(F,T,T,T,T)]
[1] 2 3 4 5
```

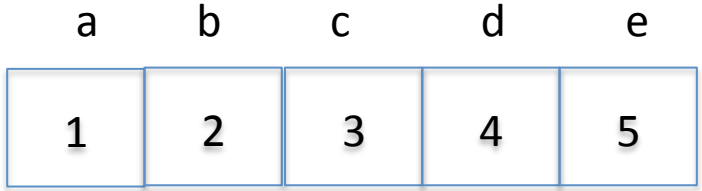
```
> x
[1] 1 2 3 4 5
>
> x[c(T,F)]
[1] 1 3 5
```

Logical Vectors - Advantages

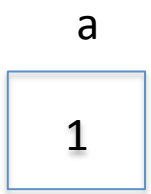
Expressions can be used to create a logical vector

```
> x
[1] 1 2 3 4 5
> b <- x < median(x)
>
> b
[1]  TRUE  TRUE FALSE FALSE FALSE
> x[b]
[1] 1 2
> x[x<median(x)]
[1] 1 2
```

(4) Character vectors to subset



Return elements with matching names




```
> x<-1:5
> x
[1] 1 2 3 4 5
> letters[x]
[1] "a" "b" "c" "d" "e"
>
> names(x)<-letters[x]
>
> x
a b c d e
1 2 3 4 5
>
> x["a"]
a
1
```

Logical Expressions in R

Operators	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	exactly equal to
!=	not equal to
!x	not x
x | y	x OR y
x & y	x AND y

```
> x
[1] 1 2 3 4 5
>
> b<- x<median(x) | x > median(x)
>
> b
[1]  TRUE  TRUE FALSE  TRUE  TRUE
>
> x[b]
[1] 1 2 4 5
```

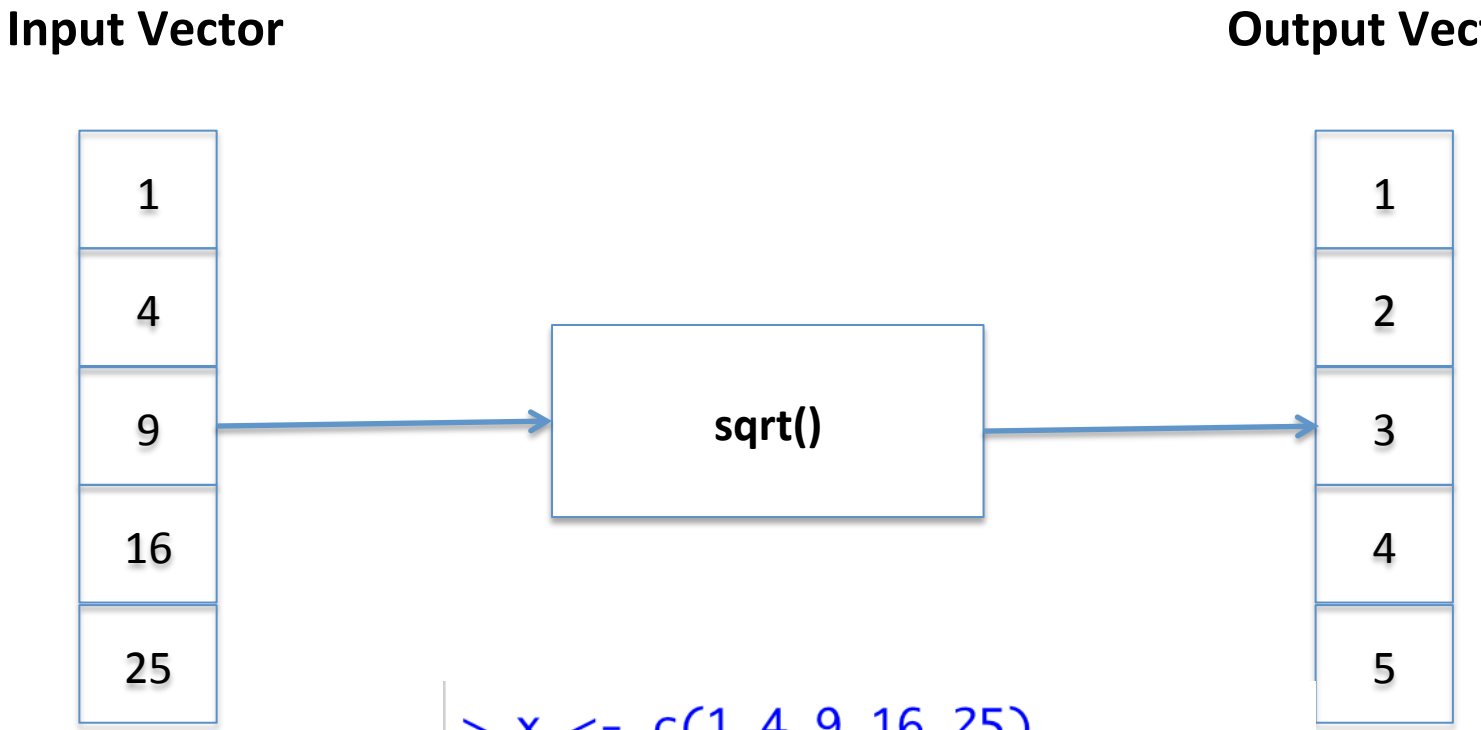
Challenge 1.2

- Create an R vector of squares of 1 to 10
- Find the minimum
- Find the maximum
- Find the average
- Subset all those values greater than the average

(4) Vectorisation

- A powerful feature of R is that it supports *vectorisation*
- Functions can operate on every element of a vector, and return the results of each individual operation in a new vector.

```
> x <- c(1,4,9,16,25)
> x
[1]  1  4  9 16 25
>
> y<- sqrt(x)
>
> y
[1] 1 2 3 4 5
```




```
> x <- c(1,4,9,16,25)
> x
[1]  1  4  9 16 25
>
> y<- sqrt(x)
>
> y
[1] 1 2 3 4 5
```

Arithmetic Operators

- Arithmetic operations can also be applied to vectors in an element-wise manner

```
> v1 <- 1:5
>
> v1
[1] 1 2 3 4 5
>
> v1 + 10
[1] 11 12 13 14 15
>
> v1 * 20
[1]  20  40  60  80 100
```

Vectorized if/else

- Vectors can also be processed using the vectorized **ifelse(b,u,v)** function, which accepts a boolean vector **b** and allocates the element-wise results to be either **u** or **v**.

```
> v1 <- 1:10
>
> v1
 [1]  1  2  3  4  5  6  7  8  9 10
>
> ifelse(v1 %% 2 == 0, "Even" , "Odd")
 [1] "Odd"  "Even" "Odd"  "Even" "Odd"  "Even" "Odd"  "Even"
 [9] "Odd"  "Even"
>
```

sample() function

`sample` takes a sample of the specified size from the elements of `x` using either with or without replacement.

Usage

```
sample(x, size, replace = FALSE, prob = NULL)

sample.int(n, size = n, replace = FALSE, prob = NULL)
```

Arguments

`x`	Either a vector of one or more elements from which to choose, or a positive integer. See 'Details.'
`n`	a positive number, the number of items to choose from. See 'Details.'
`size`	a non-negative integer giving the number of items to choose.
`replace`	Should sampling be with replacement?
`prob`	A vector of probability weights for obtaining the elements of the vector being sampled.

Example – Generate Data

- A population has 55% Female and 45% Male. Generate a vector of 50 elements to simulate this.

```
> pop<-sample(c("M","F"),50,prob = c(.45,.55),replace = T)
>
> pop
 [1] "M" "F" "F" "F" "M" "M" "F" "F" "M" "M" "F" "M" "F" "F" "M"
[16] "F" "F" "F" "F" "M" "F" "M" "F" "F" "F" "M" "F" "F" "M" "F"
[31] "F" "F" "F" "M" "F" "M" "M" "M" "M" "F" "M" "F" "F" "M" "F"
[46] "F" "M" "M" "F" "M"
>
> length(pop[pop=="M"])/50
[1] 0.42
```

Challenge 1.3

- Create a vector of random numbers between 30 and 90 (random grades for an test)
- Create a new (parallel) vector that categorises results as follows:
  - Greater or equal to 60 – Honours
  - Between 40 and 59 – Pass
  - Less than 40 – Fail
- Use the ifelse() (vectorised) function

Missing Values - NA

- In a project of any size, data is likely to be incomplete due to
  - Missed survey questions
  - Faulty equipment
  - Improperly coded data
- In R, missing data is represented by the symbol NA

```
> x <- 1:10
>
> x
 [1]  1  2  3  4  5  6  7  8  9 10
>
> x[7] <- NA
>
> x
 [1]  1  2  3  4  5  6 NA  8  9 10
>
> sum(x)
[1] NA
>
> sum(x, na.rm=T)
[1] 48
```

is.na() function

- The function is.na() indicates which elements are missing
- Returns a logical vector, the same size as the input vector

```
> x
 [1]  1  2  3  4  5  6 NA  8  9 10
>
> is.na(x)
 [1] FALSE FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
>
> x[!is.na(x)]
[1]  1  2  3  4  5  6  8  9 10
```

Challenge 1.4

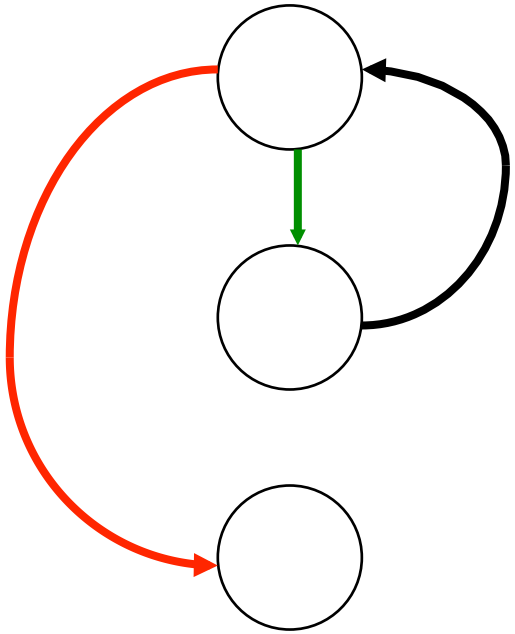
- Create a vector of 100 numbers
- Set 10 of these to NA (random)
- Print the locations of the missing values
- Hint: Check out the R function *which*

(5) Programming in R

- R is a block-structured language, where blocks are delineated by {}
- Statements separated by newline characters, or with semicolon
- Variable types are not declared (similar to JavaScript)

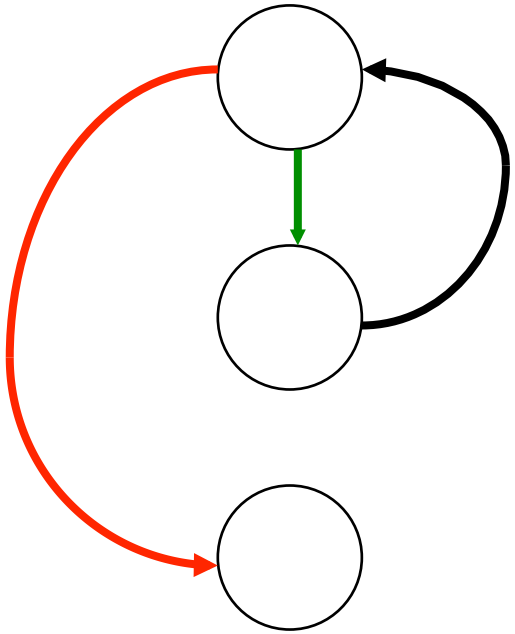
Loops - for

```
>
> x <- 1:3
>
> for(n in x){
+     print(n)
+ }
[1] 1
[1] 2
[1] 3
```



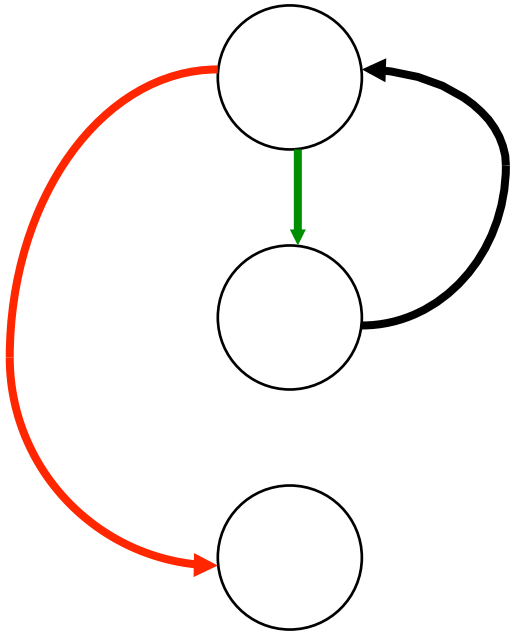
Loops - for

```
> x <- 1:3
>
> for(i in 1:length(x)){
+     print(x[i])
+ }
[1] 1
[1] 2
[1] 3
```



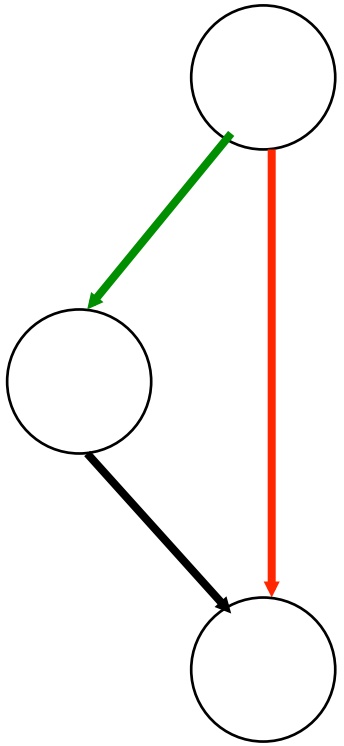
Loops - while

```
> x <- 1:3
>
> count <- 1
> while(count <= length(x)){
+     print(x[count])
+     count <- count + 1
+ }
[1] 1
[1] 2
[1] 3
```



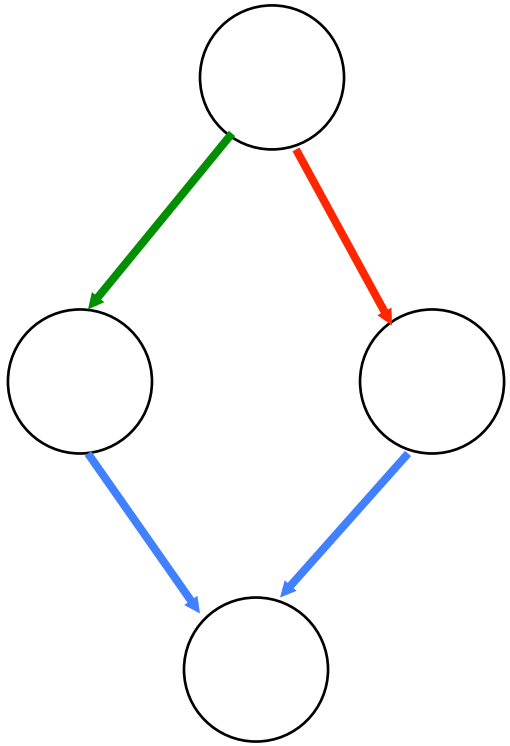
if

```
> x<-10
>
> if(x %% 2 == 0){
+    print("Even number...")
+ }
[1] "Even number..."
```



if else

```
> x<-11
> if(x %% 2 == 0){
+   print("Even number...")
+ } else
+   {
+   print("Odd number...")
+ }
[1] "Odd number..."
```



It is important to note that else must be in the same line as the closing braces of the if statements.

http://www.programiz.com/r-programming/if-else-statement

if else if

```
> x<-0
> if(x<0){
+    print("Negative number")
+ } else if (x > 0){
+    print("Positive number")
+ } else {
+    print("Zero!")
+ }
[1] "Zero!"
```

Lecture 1 Summary

- Introduction to R
- Atomic Vectors
  - Creation
  - Subsetting
  - Vectorisation
- Overview of Programming Structures (loops & ifs)

CT5102: Programing for Data Analytics

Assignment 1: Atomic Vectors

The aim of this assignment is to gain familiarity with creating and processing atomic vectors in R.

There are three tasks:

(1) Simulate the roll of two dice (N=1000), and count the number of outcomes that are odd and even. Produce the following vector as output (results should be the same as this, given that the set.seed(99) function is called.

```
> ans
 Number Odd Number Even
        523          477
```

(2) For the data generated, calculate the frequency of each outcome (i.e. the sum of the two dice values for each roll). The following vector should be generated, and the R function table() cannot be used:

```
> ans1
  2   3   4   5   6   7   8   9  10  11  12
 28  72  84 108 128 162 123 127  86  54  28
```

(3) Generate 5 random passwords of length 10, where a password can have any digit, or letter (upper and lower case). The vector to sample from should be:

```
> both
"a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q"
"r" "s" "t" "u" "v" "w" "x" "y" "z" "A" "B" "C" "D" "E" "F" "G" "H"
"I" "J" "K" "L" "M" "N" "O" "P" "Q" "R" "S" "T" "U" "V" "W" "X" "Y"
"Z" "0" "1" "2" "3" "4" "5" "6" "7" "8" "9"
```

The test output (using the seed value of 99) should be:

```
> ans2
[1] "wyyEexjOjZ" "rsqvWEOIMU" "BLWZ3MNLdR" "G1Xp7ZimvO" "yUHUrgR57Y"
```

The R Functions to be used include:

- **`set.seed(99)`**, to ensure that the results shown below are replicated.
- **`sample()`** – to generate the random samples
- **`paste()`** – for collapsing a vector of characters into a single string.
- The R constants **LETTERS** and **letters**.

Good Programming Style

http://adv-r.had.co.nz/Style.html

Style guide

Good coding style is like using correct punctuation. You can manage without it, but it sure makes things easier to read. As with styles of punctuation, there are many possible variations. The following guide describes the style that I use (in this book and elsewhere). It is based on Google's R style guide, with a few tweaks. You don't have to use my style, but you really should use a consistent style.

Good style is important because while your code only has one author, it'll usually have multiple readers. This is especially true when you're writing code with others. In that case, it's a good idea to agree on a common style up-front. Since no style is strictly better than another, working with others may mean that you'll need to sacrifice some preferred aspects of your style.

The formatR package, by Yihui Xie, makes it easier to clean up poorly formatted code. It can't do everything, but it can quickly get your code from terrible to pretty good. Make sure to read the introduction before using it.

Notation and naming

File names

File names should be meaningful and end in `.R`.

```
# Good
fit-models.R
utility-functions.R

# Bad
foo.r
stuff.r
```

References

- Lantz, B. *2013. Machine learning with R*. Packt Publishing Ltd.
- Chambers, J. 2008. Software for data analysis: programming with R. Springer Science & Business Media. Chicago.
- Duggan, J. 2016. System Dynamics Modeling with R. Springer.
- Wickham, H and Grolemund, G. 2017. R for Data Science. O'Reilly Press.
- Wickham, H. 2015. Advanced R. Taylor & Francis