Basic R

Files	
Notes	
Status	
Topics	Data Camp, Week 1

Lecture Notes

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/840134a9-4af0-4de0-9d8f-222605f46e49/01_r_programming_(Annotated).pdf

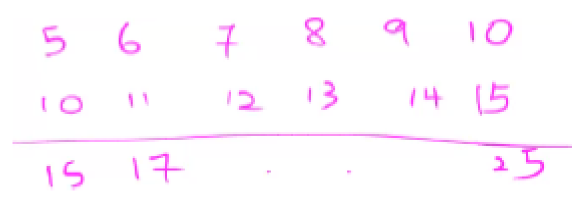
Basic Operations

Arithmetic

- Addition : `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Exponential: `^`
- Modulo: `%%`
- `?Arithmetic` - More arithmetic operations

Vectorised Operations

- Operations in `R` are all *vectorised*



If we have 2 vectors and we add them together, they will be added elements wise

The Recycling Rule

- `R` uses a recycling rule whenever it is presented with vectors of varying lengths in an expression
- `R` will recycle the shorter vector as often as needed until their lengths match. The length of the shorter vector needs to be a multiple of the longer vector
- If the vectors are not of the same multiple, they will force recycle and cuts off at the next multiple and gives a warning
- A single number is repeated an appropriate number of times

Commenting

- `#` - To add comments

Variable Assignment

- `var <- value`

Note that we use the assignment operator `<-` as compared to `=`

Keyboard shortcut for `<-` : `alt -`

▼ **File Manipulation**

- Working Directory: `getwd()`

▼ **Listing out files:**

- In the current directory: `list.files()`
- Up 1 level: `list.files("../")`
- Any directory: `list.files("filepath")`

▼ **Matrix Manipulation**

- Matrix Multiplication : `x%*%y`
- Matrix Inversion : `solve(x)`
- Matrix Transposition : `t(x)`

▼ **Working Directory**

▼ `getwd()`

- Gives the current working directory

▼ `setwd()`

- Sets the current working directory

▼ `list.files()`

- List out the files in the relevant directory stated

```
list.files() # All the files in the current working directory
list.files("../../") #Goes up 2 Folders
list.files("../data/") # this is the code that should be present in all your scripts submitted.
list.files("/home/viknesh/NUS/coursesTaught/dsa2101/src")
list.files(pattern = "*tex$") #For files that end with tex
file_list <- list.files(pattern="*tex$")
```

▼ **Getting Help**

▼ `?function`

- Shows the information about the given function

▼ `??function`

- Returns a list of search results based on the word given

▼ `example(package)`

- Runs through the examples under the help page of the package

▼ `vignette()`

- Gives a brief description of a topic

▼ **Useful Shortcuts**

▼ `Tab` key

- Code completion
- Gives a list of suggested commands that begins with the given word that has already been typed

▼ `alt` `-` Combination

- Shortcut for `->` which is useful when we want to do assignment

▼ **Packages**

▼ `install.packages("package_name")`

- Installs the listed package

▼ `library(package)`

- Loads the given `package` so that we can use the functions in the package

▼ `help(package = "package_name")`

- Access the list of all available functions from the package

▼ `search()`

- Lists out all the current packages that are loaded

▼ Sampling

▼ `sample(vector, size = n, replace = )`

- Helps to choose a sample of size `n`
- `replace` can be set to either `TRUE` or `FALSE` to state whether we want to sample with or without replacement
  - `TRUE` - Sample with replacement which means that we can have repeated values
  - `FALSE` - Sample without replacement which means that we cannot have repeated values

Data Objects

▼ Classes

▼ `character`

- Each of the element is a character string

▼ `numeric`

- Each element is a real number (e.g. those with decimals)

▼ `integer`

- Each element is an integer

▼ `logical`

- Each element is either `TRUE` or `FALSE`

▼ `factor`

- Each element is one of a few possible values. This is typically used to store categorical data

▼ `class(object)`

- Tells the `class` of the object

▼ Vectors

▼ Creation

▼ `<-` Assignment Operator

```
-2:2                # creates an evenly spaced sequence.
X <- -2:2           # assign name X to sequence
Y <- 2^X            # raise 2 to powers given by X
Y                   # print Y
length(Y)           # gives the length of the vector
c(1, 2, 3)          # creates a vector with the elements (1, 2, 3)
seq(start, end, by = ) # Note that it includes the start, end indices
```

▼ Direct assignment :

- `x <- c(1, 2, 3, 4)`

▼ From existing vectors:

- `x <- c(y, z)`

▼ From existing matrix:

- `x <- A[1,] or  x <- A[,1]`

▼ Sequences :

- `seq(start, end, by = , length = )`
  - `by =` - Step from each of the values from start to end
  - `length =` - The number of elements that we want from `start` to `end` and the separation will be decided. It could be decimals

▼ Named Vectors

        - `names(vector) <- name_vector`
        - Gives the listed `vector` names for each of the elements which we can call during the extraction

```
#Creating named vectors
names(Y) <- letters[1:5] # Assign the names of the vectors
Y <- c(a = 1, b = 2, c =3) # Assign the names of the vector during creation
```

    - ▼ `sum(vector)`
        - Gives the sum of the stated vector

- ▼ **Extraction**
    - ▼ `[]`
        - Makes use of the square brackets to indicate the relevant indices for the vector

```
Y[2]          # Access element 2
Y[2:4]        # Access elements 2,3 and 4
Y[length(Y)]  # Access the last element
Y[-1]         # Drops the first element
Y[c(1, 2,3)]  # Access index using vectors
Y[c("a")]     # Access using the names of the vector

# Q2: Suppose I want all except the last. Is this correct? Y[1:length(Y)-1]
Y[1:length(Y)-1] # Colon gets executed first and will minus off therefore the answer would be wrong
Y[1:(length(Y)-1)] #This is correct

# Q3: How to get the first and last?
Y[c(1, length(Y))]
```

        - Individually : `x[index]`
        - Collectively : `x[index_start : index_end] or x [c(1, 2, 3)] or x[-c(1, 2)]`
        - ▼ `rev(vector)`
            - Reverses the order of the vector

Note:

- Index for R starts with 1 and **not 0**
- If we put `-` before the index that we want to extract, we will extract all values other than the stated indices

- ▼ **Reversal**
    - ▼ `rev(vector)`
        - Gives a reversed form of all the elements in the given `vector`

▼ **Matrix**

- The columns must be either all be `numeric` or `character`
- ▼ **Creation**
    - Direct assignment: `A <- matrix(c(1, 2, 3, 4), ncol = ? , byrow = TRUE/FALSE)`
    - Combination of vectors: `A <-cbind(x, y) or A = rbind(x, y)`
    - ▼ `dim(matrix)`
        - Gives the dimensions of the matrix in `row, column`
    - ▼ `cbind(vector1, vector2, ...)`
        - Column wise combination
    - ▼ `rbind(vector1, vector, ...)`
        - Row wise combination
    - ▼ **Named Matrix**
        - ▼ `rownames(matrix) ← name vector`
            - Assigns the rows of the `matrix` with names that we can reference to
        - ▼ `colnames(matrix) ← name vector`

- Assigns the cols of the `matrix` with names that we can reference to

▼ **Note:**

- When using the `matrix` function, we can just state the number of columns and set either `byrow` to `TRUE` which starts putting in each of the values row by row and setting to `FALSE` will start putting in each of the values column by column
- Matrices are column majors and it will assign it column wise if no argument is passed for `byrow`
- If we were to reassign an element in a `numeric` matrix to be a `character`, it will recast the whole matrix into a `character` matrix
- If the size of the vector that is not satisfied, the vector will be recycled according to the *recycling rule* to fit the size of the matrix that is desired

▼ **Extraction**

▼ `[row, column]`

- Extraction process is according to the rows and columns that are specified

▼ **Individually :**

- `A[row, column]`

▼ **Vector :**

- `A[row, ] or A[, column]`

▼ **Matrix :**

- `A[row_vector, column_vector] or A[-row_vector, -column_vector]`

▼ **Name accessing:**

- `A[row$name, col$name]`

Note:

- If we put `-` for the rows or columns that we want to extract, we will extract parts of the matrix that does not contain those rows and columns
- If we leave the row or column empty, we will extract all the rows or columns

▼ **Data Frames**

- The columns of a data frame can have different types
- Each column is also assigned a name and all the columns have to be of the same `length`

▼ **Creation**

- **Direct Assignment:**

▼ `A <- data.frame(col_vec1, col_vec2, ...)`

- Each of the vector that is passed in will be concatenated together

▼ `B.dat <- data.frame(age = x1, income = x2)`

- For this example, x1 is assigned to the column age and x2 is assigned to the column income

- **From a .txt file:**

`read.table(file_path, header = TRUE/FALSE)`

- **From a .csv file:**

`read.csv(file_path, header = TRUE/FALSE)`

▼ **Extraction**

▼ By Name:

- `name_data_frame$column`
- Using the `$` to specify which column we want to extract

▼ By Index:

- Similar syntax to extraction as matrices
- `name_data_frame[rows, columns]`

▼ `attach(data_frame_name)`

- Attaches the data frame to the current path and we can directly call the columns without calling the data frame

▼ `detach(data_frame_name)`

- Detaches the data frame from the current path that we have previously added

▼ Lists

- Collection of objects
- Compared to data frames, the difference is the length of the items that are being stored together. We can store arbitrary items together

▼ Creation

▼ `list(object1_name = object1, object2_name = object2, ...)`

- We do not need to use the assignment operator when we are feeding the `object_name` with the `object` argument because we are not creating an object

▼ Converting to Vector

`unlist(list_object)`

We can make use of the `unlist` function to make the `list_object` a vector

▼ Extraction

- We can use either the `[]` or `[[]]` notation to extract elements

▼ `[ ]`

- Extracts only a list of the stated objects

```
#List Creation
ls1 <- list(A=seq(1, 5, by=2), B=seq(1, 5, length=4))

# extraction of the list containg only vector B
ls1["B"]

class(ls1["B"]) #list
```

▼ `list[vectors]`

- Extracts the objects in the indices or names and outputs as a

```
ls2 <- list(A=seq(1, 5, by=2), B=seq(1, 5, length=4), C=LETTERS[1:11],
            D=c(TRUE, FALSE, TRUE), E=ls1)

# Extracting the 5th and 2nd element of ls2 as a list
ls2[c(5,2)]

# Output is a list containing both items
$E
$E$A
[1] 1 3 5

$E$B
[1] 1.000000 2.333333 3.666667 5.000000


$B
[1] 1.000000 2.333333 3.666667 5.000000
```

▼ `[[ ]]`

- Extracts the contents of the stated object
- We can make use of the `name` of the object, `index` of the object in the list or `$` sign as per `matrices` or `dataframes`

```
#List Creation
ls1 <- list(A=seq(1, 5, by=2), B=seq(1, 5, length=4))

# extraction of the vector B, all of the following expressions are equivalent
ls1[["B"]]
ls1[[2]]
ls1$B

class(ls1[["B"]]) #numeric
```

▼ **Recursive Indexing**

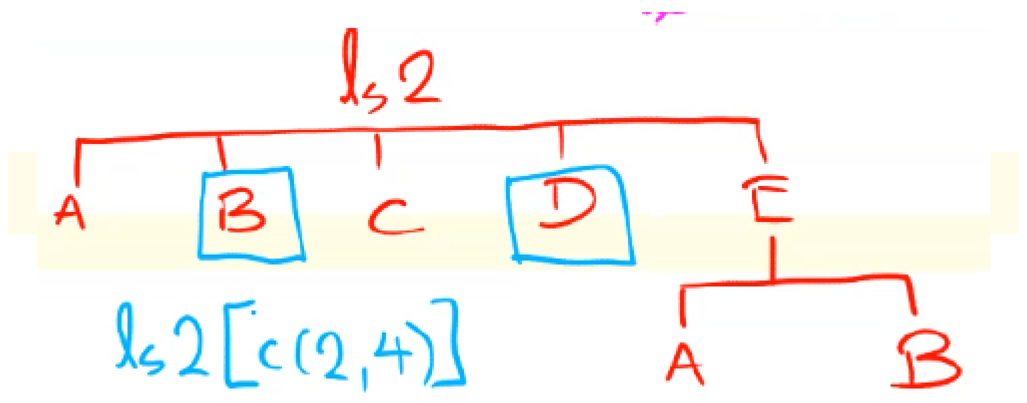
- We can nest the indices that we want to extract within the `[[ ]]` to extract the specific elements that we want

```
ls2 <- list(A=seq(1, 5, by=2), B=seq(1, 5, length=4), C=LETTERS[1:11],
            D=c(TRUE, FALSE, TRUE), E=ls1)
ls2[[c(5,1,3)]] # Does a recursive indexing (E -> A -> 3rd index) which gives 5

#Examples
# Q1: Get the "B" vector from ls1 within ls2
ls2[[c(5, 2)]]
ls2[[5]][[2]]
ls2$E$B
ls2[c(5,2)] # NOT THE SAME (WRONG)
# Q2: Get elements B and D
ls2[c(2, 4)]
ls2[c('B', 'D')]
# Q3: Drop the second element from D
ls2$D <- ls2$D[-2]
```

▼ **Note:**

- This is different as compared to the single `[]` . And it helps us to access deeper into the elements



Single `[]` will give the 2nd and 4th object in a list

▼ **Reading and Saving of Objects**

▼ **Removing Objects**

▼ **Removal of selected objects**

- `rm(list = c("object1", "object2" ...))`
  - Removes the variables that are in the stated `list`
- We can go under `Environment` tab and inspect variables and also remove them in `Grid` view

▼ **Removal of all objects**

- `rm(list = ls())`
  - Be careful and ensure that all the variables are no longer needed or saved before using this

▼ **Investigating Structure of an Object**

- `str(object, max.level = )`
  - It provides the structure of the current `object`
  - `max.level` states the maximum depth that we want to go into for the `object`
- This is especially useful when the `object` is nested and we can check it layer by layer

Conditional Execution and Loops

- **Logical Expressions**
  - **Data Types**
    - **Characters -** They will compare by the alphabetical order and capital letters are larger
    - **Numeric**
    - **Logical Values -** True is greater than False since True has a value of 1 and False has a value of 0
  - **Conditions**
    - These will yield either a `TRUE` or `FALSE` result and it will compare the values pairwise generates a logical vector
    - The **recycling rule** is in play for all of the expressions below
    - `x < y` : Lesser than
    - `x <= y` : Lesser than or equals
    - `x > y` : Greater than
    - `x >= y` : Greater than or equals
    - `x == y` : Equals
    - `x != y` : Not Equals
  - **Unary and Binary Operators**
    - `!`
      - Not operator (Element Wise)
      - Takes the negation of all the elements in the vector (Reverses it)
    - `|`
      - Or operator (Element Wise)
      - Takes the elementwise `OR` operation of two vectors
    - `||`
      - Or operator (Scalar Wise)
      - Takes the `OR` operation of two scalars, if it is a vector, it will take the element at the first index
    - `&`
      - And operator (Element Wise)
      - Takes the elementwise `AND` operation of two vectors
    - `&&`
      - And operator (Scalar Wise)
      - Takes the `AND` operation of two scalars, if it is a vector, it will take the element at the first index
  - **Selection with Logical Expressions**
    - When we have a certain criteria that is meet, it will be `TRUE` and when we pass in the logical vector into the index, it will return only those that is `TRUE` for the logical vectors

```
x <- 1:5 # Creation of the vector x
y <- x <= 3 # Gives a logical vector of [TRUE, TRUE, TRUE, FALSE, FALSE]
x[y] # Gives the elements that corresponds to the indices with TRUE which is [1, 2, 3]
```

- `if-else`
  - **Syntax:**
    1. `if (<condition>) {<result1>} elseif {<result2>} else{<result3>`
  - **Note:**

1. if the condition that is input in the **condition** in **()** is **TRUE**, it will run the first result **{<result1>}**. If not, else It will run **{<result 2>}**
2. We can use the logical operators '**&&**' & '**||**' during the execution of our if-else statement as well.
   - E.g. **if (length(x) == length(y) && x==y) {z<-x+y} else {z<-x}**

▼ `for`

▼ **Syntax:**

- `for (<condition>) {<result>}`

▼ **Note:**

1. `for` is used when we want to perform similar operations a number of times.
2. The condition is written in the form of `<variable>` in `<range>` where for each variable in the range given, the `<result>` will be executed.

▼ **Example:**

For e.g. to find the summation of 5.

```
alpha_n <- 0
n<- 5
for (i in 1:n) {alpha_n<- alpha_n+i}


# we can also iterate through the whole vector as well
test <- c(1, 2, 3)
for (item in test){
  print(item)
}
```

The above example will loop through 5 times as **i** in the **range of 1 to 5**. Thereafter, for each iteration, it will add the current **alpha_n** with the iteration of the **i** to the current **alpha_n**.

▼ `while`

▼ **Syntax**

- `while (<condition>) {<result>}`

▼ **Note:**

1. `while` is used when we want to repeat similar operations while a condition is still true
2. So long as the condition stays true, the result will continue running

▼ **Example:**

```
alpha_n <- 0
i<- 1
while (i <= 5) {
alpha_n <- alpha_n + i
i <- i + 1
}
```

▼ `repeat`

▼ **Syntax:**

- `repeat(<expression>)`

▼ **Note:**

1. `repeat` is used when we want to repeat similar operations until a condition is true
2. **Difference between repeat and while:**
   a. **`repeat`** will at least run **once** as the condition will only be checked after the first expression.
   b. **`repeat`** is used to repeat operations until a condition is met, whereas **`while`** is used to repeat operation while a condition is met

▼ `break`

- Breaks out of a loop once a condition is met (`for` & `while`)

▼ **Example:**

```
i<- 1
repeat {
  if (i==5) {break;}
  print(i);
  i <- i+1;
}
```

▼ `next`

- If there is a certain condition that is met, we can choose to skip through this current iteration and move on to the next iteration

```
x <- c(1, 2, 3)
for (i in x){
  if (i == 1){ # when i == 1, it will move on to the next iteration without executing the bottom statement
    next
  }
  print(i)
}
```

▼ `%in%`

- Checks whether the given variable is within the given vector

```
# example
if(total_val %in% c(7, 11)){
  outcome <- "win"
}

# equivalent expression
if(total_val == 7 || total_val == 11){
  outcome <- "win"
}
```

▼ `seq`

- `seq(start, end, by = , length = )`
  - `by =` - Step from each of the values from start to end
  - `length =` - The number of elements that we want from `start` to `end` and the separation will be decided. It could be decimals

▼ `seq_along()`

- Gives a sequence of integers from `1:length(vectors)` that was passed in

```
seq_along(res1)
1:length(res1)
```

▼ `do.call()`

- Constructs and executes a function call from a name or a function and a list of arguments to be passed to it
- `what` - Function that needs to be called
- `args` - List that contains all the arguments that we want to pass in, each of the items in the list will be a separate argument that gets passed into the function

```
> x <- list(c(1, 2, 3), c(3, 5, 6))
> x
[[1]]
[1] 1 2 3

[[2]]
[1] 3 5 6

> do.call(rbind, x) # each of the items in the list will be passed in
     [,1] [,2] [,3]
```

```
[1,]    1    2    3
[2,]    3    5    6
```

Functions

▼ Argument Matching

- When we call a function, we can either match the arguments by position or match it by the name

```
# By position
sd(vec)

# By name
sd(x = vec)
```

▼ In-built R Functions

▼ Mathematical Functions

`abs()` - Gives the absolute value of the numbers

`round(x, digits=0)` - Rounds the number to the nearest whole number by default

`digits` - We can specify the number of decimal places that we want in our result

`signif(x, digits = 6)` - Rounds the values in its first argument to the specified number of significant digits.

`digits` - We can specify the number of significant figures that we want for our result

`sum()` - Sums up all the number in the vector

`mean()` - Takes the mean of the numbers that are provided in the argument

`ceiling()` - Takes a single numeric argument x and returns a numeric vector containing the smallest integers not less than the corresponding elements of x.

`floor()` - Takes a single numeric argument x and returns a numeric vector containing the largest integers not greater than the corresponding elements of x.

`diff()` - Takes the difference between consecutive terms in the vector

`x` - a numeric vector or matrix containing the values to be differenced

`lag` - The number of terms to skip to take the difference between

```
x <- c(1, 2, 3, 4)

diff(x, lag = 2)
[1] 2 2
# This takes (3-1) since it skips 2 terms and also
# (4 - 2) for the next term and it will end there since for the 3rd term we do not have the 5th term
```

`differences` - The order of the difference that we want to apply

```
x <- c(10, 30, 70, 100)

diff(x, differences = 2)
[1]  20 -10
# This is actually the same as
diff(diff(x))
# The first difference yields
[1] 20 40 30
# If differences = 2, we take the difference on this vector again which results in
[1]  20 -10
```

▼ Functions for Data Structures

▼ `sort()`

- Sort (or order) a vector or factor (partially) into ascending or descending order. For ordering along more than one variable, e.g., for sorting data frames

- Note that the default would be in ascending order, we can change it to descending order by setting `decreasing = TRUE`

▼ `rep()`

- `rep(vec, times = , each = )`
  - `vec` - The vector that we want to repeat
  - `times` - The number of times we want the sequence to repeat or we can pass in a vector to indicate how many of each elements we want to repeat
  - `each` - Repeats each element the stated number of times

1. Repeat a sequence a specified number of times

```
x <- seq(1, 2, length=3) # 1.0, 1.5, 2.0
rep(x, times=3) # 1.0 1.5 2.0 1.0 1.5 2.0 1.0 1.5 2.0
```

2. Repeat each element in a sequence a specified number of times

```
x <- seq(1, 2, length=3) # 1.0, 1.5, 2.0
rep(x, times=c(1,2,3)) # 1.0 1.5 1.5 2.0 2.0 2.0
```

▼ `paste()`

▼ **Real Life Usage**

1. Used when we wish to create labels, either in a plot or in a dataset
2. When we are cleaning up data

▼ **Use**

- Used to concatenate vectors after converting them to characters
- gives a return value of the character. Does not process special characters ("\n")
- `paste(vec1, vec2, sep="")`
  - We can specify the separator under `sep` to determine what is between the 2 elements when we concatenate them
  - The concatenation will be done element-wise

```
paste ("A", 1:6 , sep = "")
# [1] "A1" "A2" "A3" "A4" "A5" "A6"

paste (c("A","B"), 1:3 , sep = "") # note that the recyling rule applies here
# [1] "A1" "B2" "A3"
```

▼ `cat()`

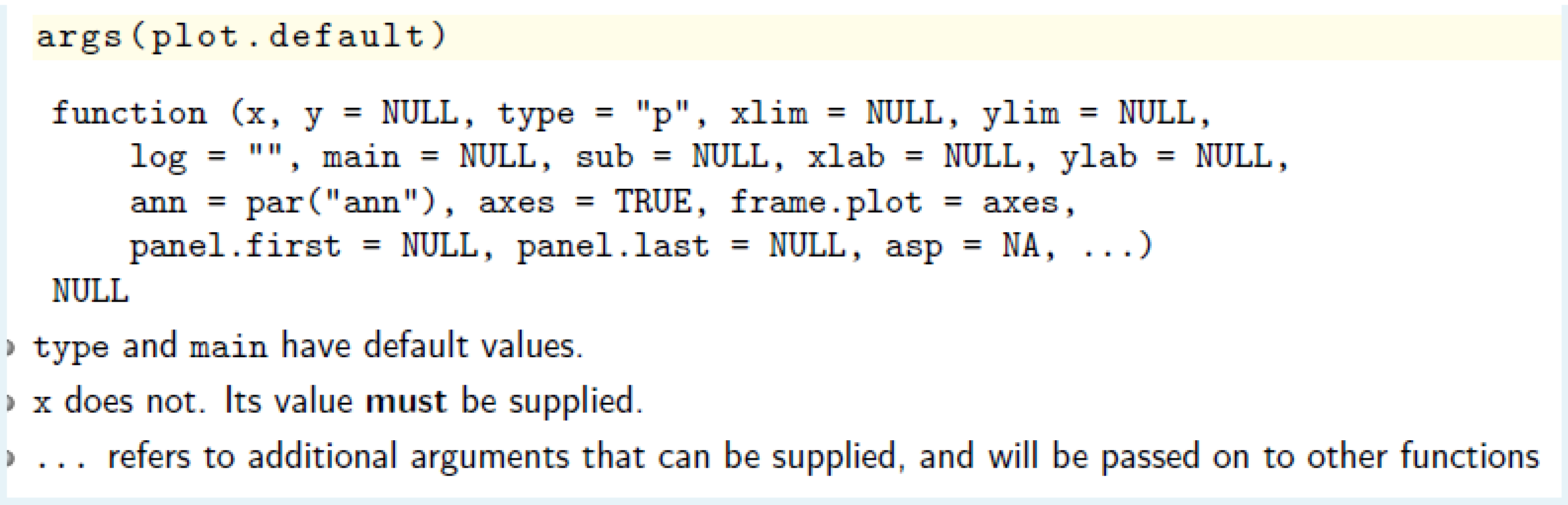
- does not have a return value and returns NULL
- helps to concat and print at the same time and it processes special characters ("\n")
- Used for printing values, we can use it with paste where paste can create the return values

▼ `str()`

- Helps to check the structure of the data

▼ `args()`

- List the arguments of a function



```
args(plot.default)
```

```
function (x, y = NULL, type = "p", xlim = NULL, ylim = NULL,
    log = "", main = NULL, sub = NULL, xlab = NULL, ylab = NULL,
    ann = par("ann"), axes = TRUE, frame.plot = axes,
    panel.first = NULL, panel.last = NULL, asp = NA, ...)
NULL
```

- `type` and `main` have default values.
- `x` does not. Its value **must** be supplied.
- `...` refers to additional arguments that can be supplied, and will be passed on to other functions

- `anyDuplicated()`
  - an integer or real vector of length one with value the 1-based index of the first duplicate if any, otherwise 0.
- `duplicated()`
  - Gives a logical vector of the positions in which the elements are duplicated. It will only take into account the first instance it happens
- `identical()`
  - The safe and reliable way to test two objects for being exactly equal. It returns `TRUE` in this case, `FALSE` in every other case.
  - `identical(x, y)` - We just need to pass in the 2 objects to check if they are exactly equal
- `is.na()`
  - Returns a logical vector of which are the positions that contains `NA` values
- `NROW()`
  - Will return the number of rows that are in the data frame
  - It will return `0` instead of `NULL` if there are no rows as compared to `nrow()`
- `is.*()`
  - They are a set of functions that allows us to check if the object is of a certain class and we just need to change `*` into the class that we want to check
- `as.*()`
  - They are a set of functions that allows us to convert the object into other classes and we just need to change the `*` into the class that we want to change into
- `unlist()`
  - Unpacks a list and changes it into a vector
  - It will coerce them into the easiest type that fits all and it could just turn them all into characters if there are of multiple types since characters can represent everything
- `rev()`
  - Reverse the order of the given argument
- `append()`
  - Adds elements to a vector
  - `x` - The vector that we want to append our values to
  - `values` - Values that we want to add to our vector
  - `after = length(x)` - We can specify the index that we want to add the values to. Note that this does not remove any of the values that are after the index and it just squeezes the additional values in, the default value is to put the values at the end of the vector
- `seq()`
  - Generate sequences, by specifying the `from`, `to`, and `by` arguments.
- `sprintf()`
  - Helps to print a formatted string

Writing R Functions

▼ `function()`

- We can make use of the `function()` keyword when we are creating our function

```
my_func <- function(args){

# Body

}
```

- When we call the function, we have to call the function with the relevant arguments passed in and call it with `( )`

▼ **Anonymous Functions**

- `function(arg) body`
- We can just define functions in the way of `function(arg) body` without any parenthesis. This kind of functions are 1 use cases

```
apply (X, 2, function (x) sum(x > 0)/ nrow (X)) # making use of anonymous function to count the proportion of values greater th
an 0

# [1] 0.81 0.85
```

▼ **Vectorising Personal Functions**

- `Vectorize()` - Wrapper functions that allows us to let our function be vectorised
    - `USE.NAMES` - Whether we want the arguments to be stated by the names or whether using the position of the argument is sufficient

▼ **Debugging**

▼ **Ways:**

1. Inserting print statements in the function to keep track
2. Inserting a breakpoint in the function, then stepping through from that point
    - `browser()` can be added into the function to create a breakpoint to enter into the code line by line to debug
3. By stepping through the function from start till finish
    - `debug()` and `undebug()` can be used to debug the function from start till finish

```
debug(my_function) # pass the function as an argument inside
my_function() # call the function to initiate the debugging
# Once we are done, we can stop the debugging from running everytime we call the function
undebug(craps_game)
```

▼ **Shortcuts:**

- `n` or `Enter` to go the next line
- `Q` to quit the debugger

▼ **`apply` Family of Functions (Built-in)**

- This family of functions allow us to repeatedly apply a function, to decide what functions that we use, it will depend on the output that we are getting

▼ **When to use `s/lapply` , `for` or neither**

- `sapply` or `lapply` - If you are repeating a function of your own
- **Neither** - If you are repeating a base R function over a vector, since these functions are already vectorized

```
# for loop
X <- sample (100 , size =50)
for(i in 1:50) {
logX [i] <- log(X[i])
}

# sapply
```

```
X <- sample (100 , size =50)
logX <- sapply (X, log)

# correct way
logX <- log(X) # because the log function is already vectorised so it takes care of the vector
```

- `for`
    - If there are many statements, and I only need to execute the loop once
    - The output from a previous iteration is required for subsequent iterations

Functions

`apply()`

- Use this if we have a have and want to apply a function to each row or column separately

```
# setting up the matrix
set. seed (2105)
X <- rnorm (200 , mean =2, sd =2)
X <- matrix (X, nrow =100)

# column application
col_means <- apply (X, 2, mean )
col_means
# [1] 1.820889 2.084210

# row application
row_means <- apply (X, 1, mean )
head ( round ( row_means , digits = 2))
# [1] 2.20 3.25 1.16 2.93 3.01 1.88

# additional arguments for the function
trimmed_col_means <- apply (X, 2, mean , trim =0.1) # trim will trim away a percentage of the datset from both sides of the
vector
trimmed_col_means
# [1] 1.742469 2.151429
```

- `apply(matrix, margin, function, ...)`
    - `matrix` - The matrix we want to operate on
    - `margin` - Whether we want to operate on the rows or columns. `1` - Rows, `2` - Columns
    - `function` - The function that we want to apply on the matrix
    - `...` - Additional arguments that we may want to supply to the function if the function itself accepts additional arguments

`sapply()`

- **Try to simplify list to array**
- Use this if the output is an `array`
- Returns a `vector` or a `matrix`
    - Note that if we are unable to simplify it into a vector or matrix, it will return us a list
- `sapply(vector, function, ...)`
    - `vector` - The vector/sequence that we want to operate on
    - `function` - The function that we want to apply on the matrix
    - `USE.NAMES` - Whether we use the col names etc from the vector itself
    - `...` - Additional arguments that we may want to supply to the function if the function itself accepts additional arguments

```
# applying the anonymous function to the list to get a vector output
street_names <- sapply(hawk_116, function(x) x$ADDRESSSTREETNAME)

# the function n times rather than feeding the values of the vector as arguments
# trick for using sapply to run a function call n times
game_results <- sapply(1:10000, function(x) craps_game())
```

▼ `replicate()`

- Use this if we want to repeat a function call multiple times and store the results in an array
- `replicate(n, function())`
  - `n` - Number of times we want to repeat the function
  - `function()` - The function call that we want to repeat

```
# replicate repeats the call n number of times
set.seed(2102)
game_results <- replicate(10000, craps_game())
table(game_results)

# equivalent to calling the below function
game_results <- sapply(1:10000, function(x) craps_game()) #This tricks R into repeating the function call n times because t
he argument that is taken in is not used
```

▼ `lappy()`

- Use this if the output is ragged or non-homogeneous whereby we need a `list` to store the values
- Returns a `list` , which is necessary if the output of each function call is not a vector of the same length
- `lapply(list, function, ...)`
  - `list` - The list/sequence that we want to operate on
  - `function` - The function that we want to apply on the matrix
  - `...` - Additional arguments that we may want to supply to the function if the function itself accepts additional arguments

```
x <- list(a = 1:10, beta = exp(-3:3),
          logic = c(TRUE,FALSE,FALSE,TRUE))
lapply(x, mean)
```

▼ `vapply()`

- **Explicitly specify output format**
- It is a safer version of `sapply` in a sense that we can specify the output that we want. We can supply a template of the output that we want and it will give us an error if it is not the expected type. This provides us better control over our data output and lets us know if there is an error earlier
- `FUN.VALUE =` - Same arguments as `sapply()` just that we need to supply this additional argument which provides a template output that we want.
  - For example if our output is a string, we can put `FUN.VALUE = "A"`
  - We could specify the number of values that we expect to return and the type

    `numeric(1)` - Numeric vector of length 1

    `numeric(2)` - Numeric vector of length 2

    `character(1)` - Character vector of length 1

    `character(2)` - Character vector of length 2

▼ `tapply()`

- We will group the values of a vector according to the values in the corresponding vector of factors and apply the function with for each of the new groups

```
X <- c(1, 2, 3)
Y <- c("a", "b", "b")
tapply(X, Y, mean) # we will group the values of X with the names of Y and apply the function

# we will apply mean to a = (1) and b = (2, 3)
a   b
1.0 2.5
```

▼ `mcapply()`

▼ `mappy()`

▼ **Higher Order Functions**

▼ `Filter()`

Structure: `Filter(f, x)`

- `f` - The function that we want to apply to filter out the vector
- `x` - The vector that we want to filter

Important Classes of Objects

▼ **Strings**

▼ **Package**

- `stringr` - Helps to carry out string operations for R
- `library(stringr)` - Loads the `stringr` package

▼ **Functions**

- **Create Strings**
  - `' '` or `" "` - Can use either to create strings
- **Compute Length of String**
  - `str_length()` - Computes the length of each string. This is a vectorized operation and we can pass in a vector of strings and they will tell us the length of each string
    - **Note:** length() - Computes the length of the vector rather than the length of the string
- **Combine Strings**
  - `str_c()` - Combine strings together (Alternative to paste)

```
str_c ("x", c("a", "y"), "z", sep =',')
[1] "x,a,z" "x,y,z"
```

- **Sub setting Strings**
  - `str_sub()` - Subsets a string

```
x <- c(" apple ", " banana ", " pear ")
str_sub (x, 1, 3)
[1] "app" "ban" "pea"

str_sub (x, 1, 20)
```

- **Converting case of a string**

`str_to_upper(string, locale = "en")` - Converts the string to all upper case

`str_to_lower(string, locale = "en")` - Converts the string to all lower case

`str_to_title(string, locale = "en")` - Converts only the first letter of each word to upper case

`str_to_sentence(string, locale = "en")` - Converts only the first letter of the first word to upper case

▼ **Factors**

▼ **Usages**

1. When we fit models
2. When we make graphs
3. When we need to work with categorical variables

▼ **Functions**

- **Creation of Factors**
    - `factor()` - Creates a factor out of the vector that is passed through

```
x1 <- c(" Dec", "Apr", "Jan", "Mar", "Apr ")

x2 <- factor (x1)
x2
#[1] Dec Apr Jan Mar Apr
#Levels: Apr Dec Jan Mar # note that the ordering of the months are based on the ordering that we pass it in
```

    - `levels()` - Checks the levels that are available in the factor
        - Note that we can specify the levels that the factor can take on even if there are no values for the given level

```
x3 <- factor (x1 , levels =c(" Jan", "Feb", "Mar", "Apr",
"May", "Jun", "Jul", "Aug",
"Sep", "Oct", "Nov", "Dec "))
x3

# [1] Dec Apr Jan Mar Apr # shows what is in the current data
# Levels: Feb Jan Mar Apr May Jun Jul Aug Sep Oct Nov Dec # shows the possible levels and it follows the ordering that
we have input
```

- **Changing of levels**
    - `factor(levels, ordered, labels)`
        - `levels` - We can state the ordering of the levels using this as well and we can make use of this to state the levels of the factors even if there are no values that have the level yet
        - `ordered` - logical vector to state whether the factor is ordered
        - `labels` - Same order as the levels and it is to state what is the label for the levels and it will be stored as that value from now on
- **Converting Numeric into Factors**
    - `cut(x, breaks, ordered_result)`
        - `x` - Numeric Vector that we want to convert to factor
        - `breaks` - either a numeric vector of two or more unique cut points or a single number (greater than or equal to 2) giving the number of intervals into which x is to be cut.
        - `ordered_result` - Whether we want the result to be an ordered factor

▼ **Date Objects**

▼ **Packages**

- `lubridate` - Provides functions that we may need for different formats (e.g. `ymd, dmy`)
- `zoo`
- `xts`

▼ **Type of Objects**

1. **Date**

▼ **Contains**

- Year
- Month
- Day

- Stored internally as integers and it is the **number of days** since `1st Jan 1970` (Therefore, it is just the number of days between the current date and `1st Jan 1970`)

💡 Note that the ISO standard of storing dates is `yyyy/mm/dd` , this allows for more efficient sorting of the dates

2. **POSIXct/ POSIXt**

▼ **Contains**

- Year
- Month
- Day
- Hours
- Minutes
- Seconds

- Stored internally as integers and it is the **number of seconds** since `1st Jan 1970 00:00:00` (Therefore, it is just the number of seconds between the current date and `1st Jan 1970 00:00:00`)

▼ **Formats**

▼ **For `Date` Objects**

- `%Y` : 4-digit year (1982)
- `%y` : 2-digit year (82)
- `%m` : 2-digit month (01)
- `%d` : 2-digit day of the month (13)
- `%A` : weekday (Wednesday)
- `%a` : abbreviated weekday (Wed)
- `%B` : month (January)
- `%b` : abbreviated month (Jan)

▼ **For `POSIXct` Objects**

- `%Y` : 4-digit year (1982)
- `%y` : 2-digit year (82)
- `%m` : 2-digit month (01)
- `%d` : 2-digit day of the month (13)
- `%A` : weekday (Wednesday)
- `%a` : abbreviated weekday (Wed)
- `%B` : month (January)
- `%b` : abbreviated month (Jan)
- `%H` : hours as a decimal number (00-23)
- `%I` : hours as a decimal number (01-12)
- `%M` : minutes as a decimal number
- `%S` : seconds as a decimal number
- `%T` : shorthand notation for the typical format `%H:%M:%S`
- `%p` : AM/PM indicator

- Note that when we are putting in the format for the dates in `as.Date()` or `as.POSIXct()` we just have to follow the format of the strings that are in the vector and if there are spaces or any symbols we can just put it in, so long as it is the same as the string that is being passed in

▼ **Help**

`?strftime` - Help page for date time change

`?seq.Date` - Help page for sequence dates

Arithmetic

- Note that we can use `+ -` with the `Date` or `POSIXct` objects

Functions

`as.Date()` - Converts the given string into a `Date` object

- Note that the default format if the format is specified is the ISO standard of `YYYY-MM-DD` and it will automatically convert to that
- `as.Date("date_string", "format")`
- The different formats can be found under the help page of `?strftime`

`as.POSIXct()` - Converts the given string into a `POSIXct` object

- The default format will be in the ISO format of `YYYY-MM-DD HH:MM:SS`
- `as.POSIXct("posixct_string", "format")`

`weekdays()` - Provides the weekdays of the given vector (E.g. Monday, Tuesday)

- `weekdays(vector, abbreviate = ...)`

  `vector` - The date vector that we want to find the weekdays

  `abbreviate` - We can put it to be either `TRUE` or `FALSE` which will allow for the abbreviated versions of the weekdays

`months()` - Provides the months of the given vector (E.g. January, February)

- `months(vector, abbreviate = ...)`

  `vector` - The date vector that we want to find the months

  `abbreviate` - We can put it to be either `TRUE` or `FALSE` which will allow for the abbreviated versions of the months

`Sys.Date()` - Provides the ISO date for the current day

- `Date` object

`seq()` - Generate a sequence of dates

- `seq(from,to, by = , length.out =)`

  `from` - Starting Date (Required)

  `to` - End Date (Required)

  `by` - Increment of the sequence which can contain the following

  - A number, taken to be in days.
  - A object of class difftime
  - A character string, containing one of `"day"`, `"week"`, `"month"`, `"quarter"` or `"year"`. This can optionally be preceded by a (positive or negative) integer and a space, or followed by "s".

  `length.out` - Integer, optional. Desired length of the sequence.

`Sys.time()` - Gives the ISO Date with the time for the current day

- `POSIXct/POSIXt` Object

`cut()` - Helps to group the dates into months or weeks or quarters together

- `cut(x, breaks, labels)`

  `x` - Numeric vector which is to be converted to a factor by cutting, we can pass in our date vector here

  `breaks` - Either a numeric vector of two or more unique cut points or a single number (greater than or equal to 2) giving the number of intervals into which x is to be cut. We can pass in as the `"months" "weeks"` etc.

- `quarter` - Will break it into intervals of 3 calendar months (`beginning on January 1, April 1, July 1 or October 1`)

`labels` - If set to `FALSE`, it represents the groupings for each of the elements but note that it does not represent the month. If we use the default which is `TRUE`, it will state out the different levels as a factor

▼ **Common Issues**

▼ Converting a vector into a Date Object when there are empty elements

- For this, we could have different strings that represents that the vector `" " "N/A"` , to solve it we can change all instances of the empty elements into `NA` which stands for empty entries then we cast as a date object using `as.Date()`

```
today <- Sys.Date()
seq(today - 100, today, by="1 months")

s1 <- seq(today - 100, today, by="1 week")
s1[1:3]

s2 <- as.character(s1)
s2[sample(15, size=5)] <- c("#N/A", "", "#N/A", "#N/A", "")

# Solution
s2_tmp <- ifelse(s2 %in% c("#N/A", ""), NA, s2) # create as a character vector first
new_s2 <- as.Date(s2_tmp) # we will cast the whole vector as a date object in one shot
```

Regular Expressions

▼ **Procedure**

1. Write out the pattern that we want to match
2. After matching:
    a. Detect elements that have the pattern
    b. Replace a pattern with another string
    c. Extract what's before or after the pattern
    d. Split the string at the position of the matched pattern position

▼ **Help**

- `?about_search_regex` - Help page for regular expression

▼ **Functions**

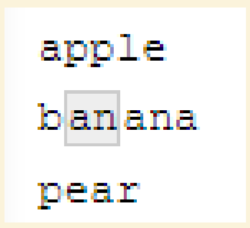
▼ **Functions without the use of `stringr`**

- **`grepl()`** **-**which returns `TRUE` when a pattern is found in the corresponding character string.
- **`grep()`** **-** which returns a vector of indices of the character strings that contains the pattern.
- `sub()` - We can specify a replacement argument. If inside the character vector x, the regular expression pattern is found, the matching element(s) will be replaced with replacement
    - This only does it for the first match
    - Note: `([0-9]+)`: The parentheses are used to make parts of the matching string available to define the replacement. The `\\1` in the replacement argument of `sub()` gets set to the string that is captured by the regular expression `[0-9]+`
- `gsub()` - We can specify a replacement argument. If inside the character vector x, the regular expression pattern is found, the matching element(s) will be replaced with replacement
    - This does it for all matches
    - Note: `([0-9]+)`: The parentheses are used to make parts of the matching string available to define the replacement. The `\\1` in the replacement argument of `sub()` gets set to the string that is captured by the regular expression `[0-9]+`

💡 Note that we will need `library(stringr)` to run the string functions

- **Testing of Regular Expressions**
  - `str_view()` - Allows for testing for regular expressions and it will give a display of which are the parts of the strings that matches
  - `str_view_all()` - It shows all the patterns in the word rather than just the first match

```
str_view(x, "an")
```



```
apple
banana
pear
```

It will show us where it matches

- **Detecting Matches**
  - `str_detect()` - It detects the match and returns a logical vector of which are the elements that contains the match
- **Replacing Matches**
  - `str_replace()` - Replaces the first instance of the match of the string

```
str_replace(vector, "pattern", "string_to_replace_with")
```

  - `str_replace_all()` - Replaces all instances of the match of the string and also works for vectorized operations

```
str_replace_all(vector, "pattern", "string_to_replace_with") # replaces all instances of the given pattern

str_replace_all(vector, c("pattern1" = "string_to_replace_with1", "pattern2" = "string_to_replace_with2")) # replaces all i
nstances of either of the given pattern
```

▼ **Note**

- We can specify specific patterns in the regex pattern with `( )` and we can use `\\1` or other indexes to replace the whole string with only that part of the string, if there are no matches, then it will not replace anything

```
awards <- c("Won 1 Oscar.",
+             "Won 1 Oscar. Another 9 wins & 24 nominations.",
+             "1 win and 2 nominations.",
+             "2 wins & 3 nominations.",
+             "Nominated for 2 Golden Globes. 1 more win & 2 nominations.",
+             "4 wins & 1 nomination.")


> str_replace( awards, ".*\\s([0-9]+)\\snomination(.*$)", "\\1") # We replace the whole string by the part that was found u
nder [0-9]+ since it is the first paranthesis
[1] "Won 1 Oscar." "24"           "2"            "3"            "2"
[6] "1"

>str_replace( awards, ".*\\s([0-9]+)\\snomination(.*$)", "\\2") # We replace the whole string by the part that was found un
der (.*$) since it is the second paranthesis
[1] "Won 1 Oscar." "s."           "s."           "s."           "s."
[6] "."
```

- **Extracting Matches**
  - `str_extract()` - Extracts the first instance of the given match in the word

```
str_extract(vector, "pattern")
```

- `str_extract_all()` - Extracts all instances of the given match in the word

  ```
  str_extract_all(vector, "pattern")
  ```

▼ Patterns

▼ Grouping

`( )` - Groups a series of pattern elements to a single element. When you match a pattern within parentheses, you can use any of the other pattern position/ number of times

`[ ]` - Denotes a set of possible character matches.

```
[Jurong] # we can match either of the characters inside here which is "J" or "u" or "r" or "o" or "n" or "g"
```

`|` - Separates alternate possibilities

```
(Hello|Hi) # we can either match the whole word "Hello" or "Hi"
```

`[^...]` - Matches every character except the ones inside brackets

▼ Position

`^` - Start of the string

`$` - End of the string

`(?=...)` - Look Ahead Assertion

- If we find a pattern that matches within `...` then the pointer will start from the position before the character and we can specify anything before this and it will match the pattern

```
x <- c("apple", "banana", "pear")
str_view(x, ".(?=e)") # This matches any character that appears before "e" in the string. This will pick up the "l" from ap
ple and "p" from pear
```

`(?<=...)` - Look Behind Assertion

- If we find a pattern that matches within `...` then the pointer will start from the position after the character and we can specify anything after this and it will match the pattern

```
x <- c("apple", "banana", "pear")
str_view(x, "(?<=e).") # This matches any character that appears after "e" in the string. This will only pick up "a" from p
ear
```

`(?!...)` - Negative Look Ahead Assertion

- If we find a pattern that **does not** matches within `...` then the pointer will start from the position before the character and we can specify anything before this and it will match the pattern. Note that if there is no value before the first match, it will go to the next character

```
x <- c("apple", "banana", "pear")
str_view(x, ".(?!a)")

apple # matches p first since it is the first non a character, then it looks ahead so it matches the a
banana # matches b first since it is the first non a character, but look ahead, there is nothing, so it goes to the next ma
tch, which is n (since a cannot match) and it looks forward so the match is a
pear # matches p first since it is the first non a character, but look ahead, there is nothing, so it goes to the next matc
h, which is e, and it looks forward so the match is p
```

`(?<!...)` - Negative Look Behind Assertion

- If we find a pattern that **does not** matches within `...` then the pointer will start from the position after the character and we can specify anything after this and it will match the pattern

```
```

▼ **Number of Times**

Greedy vs Reluctant vs Possessive

- **Place the symbols after the pattern**

▼ **Greedy** - Match as much as possible to the greedy quantifier and the entire regex. If there is no match, backtrack on the greedy quantifier. It matches even the patterns after that, then it will slowly reduce the number of matches that it takes

`*` - Matches the preceding pattern element 0 or more times.

`+` - Matches the preceding pattern element 1 or more times.

`?` - Matches the preceding pattern element 0 or 1 time.

`{n}` - Matches exactly `n` times

`{n,}` - Matches at least `n` times. Match as many times as possible

`{n, m}` - Match between `n` and `m` times. Match as many times as possible, but not more than m

▼ **Example**

```
Input String: xfooxxxxxxfoo
Regex: .*foo

The above Regex has two parts:
(i)'.*' and
(ii)'foo'

Each of the steps below will analyze the two parts. Additional comments for a match to 'Pass' or 'Fail' is explained w
ithin braces.

Step 1: # It goes all to the end for the greedy quantifier first
(i) .* = xfooxxxxxxfoo - PASS ('.*' is a greedy quantifier and will use the entire Input String)
(ii) foo = No character left to match after index 13 - FAIL
Match failed.

Step 2:
(i) .* = xfooxxxxxxfo - PASS (Backtracking on the greedy quantifier '.*')
(ii) foo = o - FAIL
Match failed.

Step 3:
(i) .* = xfooxxxxxxf - PASS (Backtracking on the greedy quantifier '.*')
(ii) foo = oo - FAIL
Match failed.

Step 4:
(i) .* = xfooxxxxxx - PASS (Backtracking on the greedy quantifier '.*')
(ii) foo = foo - PASS
Report MATCH

Result: 1 match(es)
I found the text "xfooxxxxxxfoo" starting at index 0 and ending at index 13.
```

▼ **Reluctant** - Match as little as possible to the reluctant quantifier and match the entire regex. if there is no match, add characters to the reluctant quantifier. Opposite from greedy, it starts from the least then it goes up until there is a match

`?` - Modifies the symbols to represent as few times as possible instead

`*?` - Match 0 or more times. Match as few times as possible.

`+?` - Match 1 or more times. Match as few times as possible.

`??` - Match zero or one times. Prefer zero.

`{n}?` - Match exactly n times.

`{n,}?` - Match at least n times, but no more than required for an overall pattern match.

`{n,m}?` - Match between n and m times. Match as few times as possible, but not less than n.

▼ **Example**

```
Input String: xfooxxxxxxfoo
Regex: .*?foo

The above regex has two parts:
(i) '.*?' and
(ii) 'foo'

Step 1:
.*? = '' (blank) - PASS (Match as little as possible to the reluctant quantifier '.*?'. Index 0 having '' is a match.)
foo = xfo - FAIL (Cell 0,1,2 - i.e index between 0 and 3)
Match failed.

Step 2:
.*? = x - PASS (Add characters to the reluctant quantifier '.*?'. Cell 0 having 'x' is a match.)
foo = foo - PASS
Report MATCH

Step 3:
.*? = '' (blank) - PASS (Match as little as possible to the reluctant quantifier '.*?'. Index 4 having '' is a match.)
foo = xxx - FAIL (Cell 4,5,6 - i.e index between 4 and 7)
Match failed.

Step 4:
.*? = x - PASS (Add characters to the reluctant quantifier '.*?'. Cell 4.)
foo = xxx - FAIL (Cell 5,6,7 - i.e index between 5 and 8)
Match failed.

Step 5:
.*? = xx - PASS (Add characters to the reluctant quantifier '.*?'. Cell 4 thru 5.)
foo = xxx - FAIL (Cell 6,7,8 - i.e index between 6 and 9)
Match failed.

Step 6:
.*? = xxx - PASS (Add characters to the reluctant quantifier '.*?'. Cell 4 thru 6.)
foo = xxx - FAIL (Cell 7,8,9 - i.e index between 7 and 10)
Match failed.

Step 7:
.*? = xxxx - PASS (Add characters to the reluctant quantifier '.*?'. Cell 4 thru 7.)
foo = xxf - FAIL (Cell 8,9,10 - i.e index between 8 and 11)
Match failed.

Step 8:
.*? = xxxxx - PASS (Add characters to the reluctant quantifier '.*?'. Cell 4 thru 8.)
foo = xfo - FAIL (Cell 9,10,11 - i.e index between 9 and 12)
Match failed.

Step 9:
.*? = xxxxxx - PASS (Add characters to the reluctant quantifier '.*?'. Cell 4 thru 9.)
foo = foo - PASS (Cell 10,11,12 - i.e index between 10 and 13)
Report MATCH

Step 10:
.*? = '' (blank) - PASS (Match as little as possible to the reluctant quantifier '.*?'. Index 13 is blank.)
foo = No character left to match - FAIL (There is nothing after index 13 to match)
Match failed.

Result: 2 match(es)
I found the text "xfoo" starting at index 0 and ending at index 4.
I found the text "xxxxxxfoo" starting at index 4 and ending at index 13.
```

▼ **Possessive** - Match as much as possible to the possessive quantifier and match the entire regex. Do NOT backtrack. Same as greedy but does not backtrack

`+` - Modifies the symbols to represent possessive match instead

`*+` - Match 0 or more times. Match as many times as possible when first encountered, do not retry with fewer even if overall match fails (Possessive Match).

`++` - Match 1 or more times. Possessive match.

`?+` - Match zero or one times. Possessive match.

`{n}+` - Match exactly n times.

`{n,}+` - Match at least n times, Possessive match.

`{n,m}+` - Match between n and m times. Possessive match.

▼ **Example**

```
Input String: xfooxxxxxxfoo
Regex: .*+foo

The above regex has two parts: '.*+' and 'foo'.

Step 1:
.*+ = xfooxxxxxxfoo - PASS (Match as much as possible to the possessive quantifier '.*')
foo = No character left to match - FAIL (Nothing to match after index 13)
Match failed.

Note: Backtracking is not allowed.

Result: 0 match(es)
```

▼ **Characters**

💡 Note that for these characters, we will need to use `\\...` instead so that R knows that we are dealing with these characters rather than the single `\` being an escape key

`\w` - Matches an **alphanumeric** character, including "`_`"; same as `[A-Za-z0-9_]` in ASCII, and `[\p{Alphabetic}\p{GC=Mark}\p{GC=Decimal_Number}\p{GC=Connector_Punctuation}]` in Unicode, where the `Alphabetic` property contains more than Latin letters, and the `Decimal_Number` property contains more than Arab digits.

`\W` - Matches a **non alphanumeric** character, excluding "`_`"; same as `[^A-Za-z0-9_]` in ASCII, and `[^\p{Alphabetic}\p{GC=Mark}\p{GC=Decimal_Number}\p{GC=Connector_Punctuation}]` in Unicode. (Opposite of `/w`)

`\s` - Matches a **whitespace** character, which in ASCII are tab, line feed, form feed, carriage return, and space; in Unicode, also matches no-break spaces, next line, and the variable-width spaces (amongst others).

`\S` - Matches anything that is **not** a whitespace character

`\d` - Matches a **digit**; same as `[0-9]` in ASCII

`\D` - Matches a **non-digit**; same as `[^0-9]` in ASCII

`.` - Normally matches any character except a newline.

Within square brackets the dot is literal.

`\b` - Match if the current position is a word boundary. Boundaries occur at the transitions between word (`\w`) and non-word (`\W`) characters, with combining marks ignored. For better word boundaries

`\B` - Match if the current position is not a word boundary.

`[0-9] == [0123456789]`

`[a-z] == [abcdefghijklmnopqrstuvwxyz]`

`[A-Z] == [ABCDEFGHIJKLMNOPQRSTUVWXYZ]`

Plotting

▼ **Help**

`?par` - Help page for graphical parameters for graphs

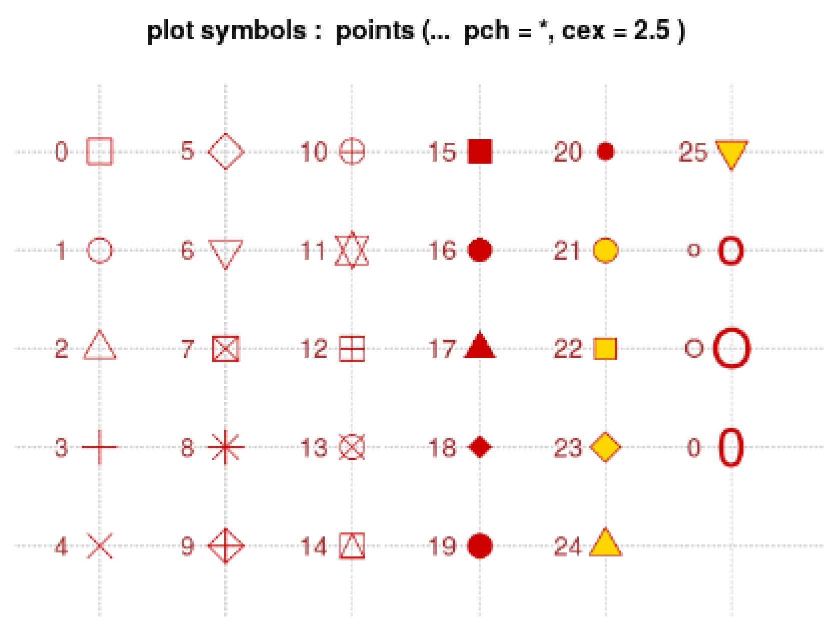
`?pch` - Help page for plotting characters for graphs

▼ **Functions:**

▼ `plot()`

- Produces a scatter plot
- `plot(x, y, main=” “, xlab = “ “, ylab” “, type = “ “, col = “ “)`
  - ▼ **Data**
    - `x` – dataset of the x-axis
    - `y` – dataset of the y-axis
    - Note that we can pass in the data as the following instead of just stating the x and y values
      - A list with components x and y, or
      - A data frame
      - A matrix with 2 columns.
  - ▼ `main` – title of the plot
  - ▼ `xlab` – label for the x-axis
  - ▼ `ylab` – label for the y-axis
  - ▼ `xlim` - Range of the x axis (needs to be a vector of length 2)
  - ▼ `ylim` - Range of the y axis (needs to be a vector of length 2)
  - ▼ `type` – plot type
    1. “ `p` ”- points
    2. “ `l` ” – lines
    3. “ `b` ” – both points and lines
    4. “ `c` ” – empty points joined by lines
    5. “ `o` ” – overplotted points and lines
    6. “ `s` ” and “S” – stair steps
    7. “ `h` ” – histogram-like vertical lines
    8. “ `n` ” – does not produce any points or lines
  - ▼ `col` – colour of the plot
    - `colours()` - States all the colours that are available
    - `rgb(r, g, b, alpha)` - Allows us to create our colours by stating the red green blue and also the transparency

      `alpha` - It gives the amount of transparency. E.g. `alpha = 0.4` will be that if there are 4 points, we will not be able to see through it anymore (This is useful if we have duplicated points)
  - ▼ `pch` - plotting characters
    - Default plotting character is `pch=1`



The type of symbols that we can plot and the numbers that correspond to them. ?pch to see more information

- ▼ `cex` - character expansion, changes the size of the elements of the graph
    - Default value is `1`
    - Larger values will make the symbol larger, smaller values will make it smaller
    - `cex.axis` - affects the font size in the axis
    - `cex.main` - affects the font size in the title
- ▼ `bty` - states the box type.
    - A character string which determined the type of box which is drawn about plots. If bty is one of "o" (the default), "l", "7", "c", "u", or "]" the resulting box resembles the corresponding upper case letter. A value of "n" suppresses the box.
- ▼ `lty` - states the line type
    - The line type. Line types can either be specified as an integer (0=blank, 1=solid (default), 2=dashed, 3=dotted, 4=dotdash, 5=longdash, 6=twodash) or as one of the character strings "blank", "solid", "dashed", "dotted", "dotdash", "longdash", or "twodash", where "blank" uses 'invisible lines' (i.e., does not draw them).

▼ `par()`

- Changes the graphical parameters of the graph
- Note that once we call `par()` it will return the previous setting and we can store in another object so that we can revert back to the old settings

▼ **Parameters**

`mfrow()` - States the number of plots by rows and columns. The plots will be added row by row

- We can pass in a vector `mfrow=c(2, 2)` whereby this is 2 rows and 2 columns but row major

`mfcow()` - States the number of plots by rows and columns. The plots will be added column by column

- We can pass in a vector `mfcow=c(2, 2)` whereby this is 2 rows and 2 columns but column major

`mar()` - States the amount of margin that we want to give `(bottom, left, top, right)`

- Default is `c(5, 4, 4, 2) + 0.1`

`dev.off()` - Goes back to the default settings

`par(original_setting_object)` - If we saved the original object, we can revert back to the original setting using this

▼ `abline()`

- Adds one or more straight lines through the current plot
- `abline(a, b, h, v)`

`a` - Y-intercept of the line

`b` - Slope of the line

`h` - the y-value(s) for horizontal line(s)

We can use this to add horizontal lines with a certain y value

`v` - the x-value(s) for vertical line(s).

We can use this to add vertical lines with a certain x value

▼ `line()`

- `line(x, y, col)`
    - `x` and `y` can be vectors of the same length and it will plot according to the different points
- Plots out a line and is added to a plot

▼ `point()`

- `point(x, y, col)`
- Plots out a range of points and is added to a plot

▼ `legend()`

- `legend("topleft", c("sin(x)", "cos(x)"), fill=c("blue", "red"))`
- Adds a legend to a plot. Able to concatenate and overlay plots.

▼ `barplot()`

- Bar charts represent a variable by drawing bars whose heights are proportional to the values of the variable.
- Produces a frequency plot with vertical or horizontal bars
- `barplot(height, border, width names.arg`)

  `height` - Either a vector or matrix of values describing the bars which make up the plot. Each column of a data.frame or matrix will be used as a plot by itself or collate them all under the same plot

  `width` - Optional vector of bar widths. Re-cycled to length the number of bars drawn. Specifying a single value will have no visible effect unless xlim is specified.

  `beside` - A logical value. If `FALSE`, the columns of height are portrayed as stacked bars, and if `TRUE` the columns are portrayed as juxtaposed bars. Can be used when there are multiple columns being passed in with a `data.frame`

  `border` - The color to be used for the border of the bars. Use border = NA to omit borders. If there are shading lines, border = TRUE means use the same colour for the border as for the shading lines.

  `args.legend` - List of additional arguments to pass to legend(); names of the list are used as argument names. Only used if legend.text is supplied.

  `names.arg` - A vector of names to be plotted below each bar or group of bars. If this argument is omitted, then the names are taken from the names attribute of height if this is a vector, or the column names if it is a matrix.

  `legend.text` - A vector of text used to construct a legend for the plot, or a logical indicating whether a legend should be included. This is only useful when height is a matrix. In that case given legend labels should correspond to the rows of height; if legend.text is true, the row names of height will be used as labels if they are non-null.

  `horiz = TRUE` - Whether to plot the bars horizontally instead of vertically

  `las` - numeric in {0, 1, 2, 3}, the style of the axis labels

    0: always parallel to the axis (default)

    1: always horizontal

    2: always perpendicular to the axis

    3: always vertical

▼ `hist()`

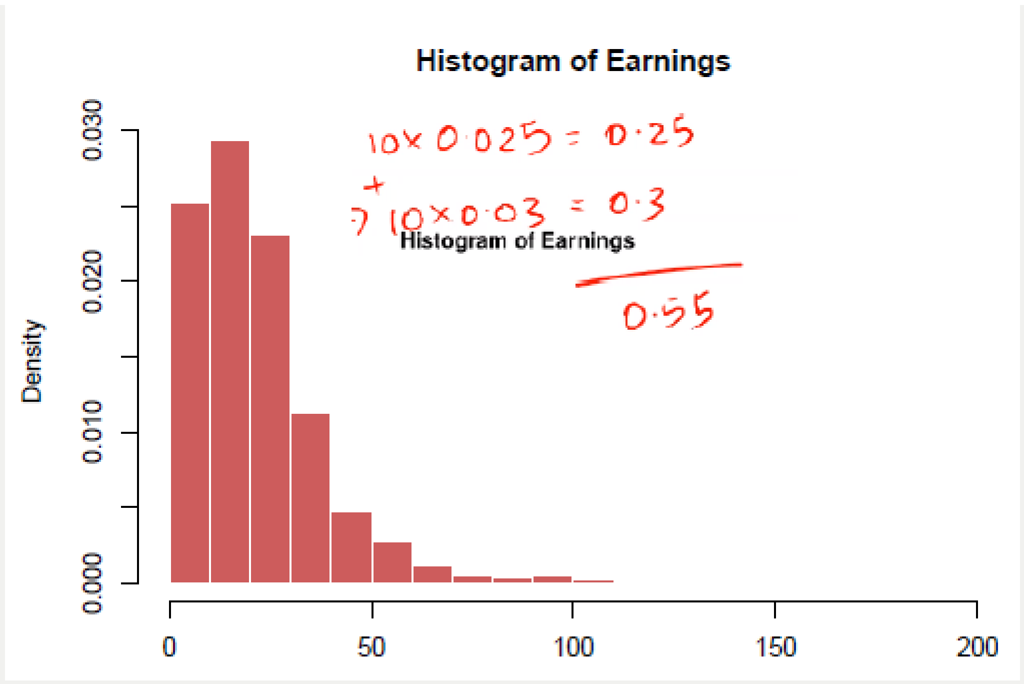
- Produces histograms, it divides the range of values into bins, then counts the number of values that fall into each bin

▼ Things to check for a histogram

1. Skewness/Symmetric
   - Right skew means that it is longer on the right side which makes the data skewed and the left side will have a lot more data
2. Unimodal or Multimodal Distribution
3. Mode
4. Check the outliers

▼ Note

- Note that histograms are different from bar charts, bar charts have categories on the x axis. Whereas for histograms, they have a continuous numerical values
- We can compute the proportion of the people having a certain amount using the area of graph if we set `freq=FALSE`



This calculation shows that 55% of the population have income less than 20 thousand per annum because we are taking the area of the first 2 bins

- `hist(x, breaks , col, main, xlab, ylab)`
  - ▼ `x` – vector of values for which the histogram is desired
  - ▼ `breaks`:

    Can be any of the following values

    1. a vector giving the breakpoints between histogram cells
    2. a function to compute the vector of breakpoints
    3. a single number giving the number of bins for the histogram
    4. a character string naming an algorithm to compute the number of cells
    5. a function to compute the number of cells
  - ▼ `col` – colour of the plot
  - ▼ `main` – title of the plot
  - ▼ `xlab` – label for the x-axis
  - ▼ `ylab` – label for the y-axis
  - ▼ `prob` – Whether the y-axis should be turned into **probability** because the y-axis is the frequency and if we turn it into probability it is just probability of the value from the x-axis happening. If **prob=T**, the overall probability for all values is **1**.
  - ▼ `freq` - Default value is `TRUE` . If set to `FALSE`, it alters the histogram such that the height of each bar does not represent a count but the probability instead. The area of the histogram will sum up to be 1 instead
  - ▼ `border` - colour of the borders
  - ▼ `na.strings` - Provides the values for empty values
  - ▼ `colClasses` - States the classes of the columns
    - We can pass in a vector with the same length as the columns and stating the classes for each of the columns

▼ `png()`

- Saves the plot in a png file in the current working directory. The argument that is passed through will be the name of the png file with the .png extension written within it.
- Once this is run, for any plot that is run, there will not be a pop-up. It will all save under the png file.
- `dev.off()` – has to be run to end the whole `png()` operation and to tell it to save.