Introduction to programming in R

Day 2

Introduction to programming in R
March 2025

Digital Research Alliance of Canada | Alliance de recherche numérique du Canada

Recap of yesterday

- We learnt about data types and data structures.

- Data types in R are **numeric**, **character**, and **logical**.

    - **Numerics** are any number (whole or decimal)

    - **Characters or strings** are any combination of letters and symbols, confined within " "

      (character is usually referred to a single letter/symbol and string refers to multiple letters/word/sentence)

    - **Logicals** are either TRUE/FALSE. These are usually outcomes of a comparison or condition:

      Is `a` larger than `b`? Is there missing data?

Recap of yesterday

- There are several ways in which we can store data in R. The most commonly used ones are **vectors, matrices, and data frames**.
    - **Vectors** are a concatenated sequence of any data type (same type).
    - **Matrices** are a table of any data type (same data type).
    - **Dataframes** are a table that allows different data types.

Recap of yesterday

- We then learnt about **operators and functions**.
- Operators are symbols or keywords that perform arithmetic, logical, relational, or assignment operations.

Relational operators

Operator	Description
`a > b`	Tests for greater than
`a < b`	Tests for smaller than
`a >= b`	Tests for greater or equal than
`a <= b`	Tests for smaller or equal than
`a == b`	Tests for equality
`a != b`	Tests for inequality

Logical operators

Operator	Description
`!`	Logical NOT
`&&`	Logical AND
`||`	Logical OR

= and <- : assignment operators

Recap of yesterday

- **Functions** are contained blocks of code, that perform a set of pre-defined tasks: `print("something")`
- **Arguments** are the values you pass to a function to control its behaviour. They help customize the function's output.

```
sum(1, 2, 3, 4)

round(3.14159, digits = 2)
```

Recap of yesterday

- We then learnt about **Conditions and loops.**
- Conditions and loops are structures that allow us to control the execution of command.
- If (If-Else) statements: they check for a condition, and execute certain tasks if the condition holds.

Anything that is type **logical** can go here.

```
if(condition) {

  Task 1
  Task 2
  …

  }
```

If the condition is **TRUE**, then the commands within the brackets are **executed**.

If the condition is **FALSE**, then the commands within the brackets are **ignored**.

Recap of yesterday

- We then learnt about **Conditions and loops.**
- Conditions and loops are structures that allow us to control the execution of command.
- Loops are structures that allow us to repeat similar actions.

```
for (variable in sequence) {
   Task1
   Task2
   Task3
}
```

Variable → A dummy variable that stores the current value of the loop in the sequence.
Sequence → A set of values the loop goes through (e.g., numbers 1 to 5, names of fruits, etc).
Loop body {Task1, Task2, Task3} → The code inside the { } runs once for each value in the sequence.

Recap of yesterday

```
for (variable in sequence) {
   Task1
   Task2
   Task3
}
```

Variable → A dummy variable that stores the current value of the loop in the sequence.
Sequence → A set of values the loop goes through (e.g., numbers 1 to 5, names of fruits, etc).
Loop body {Task1, Task2, Task3} → The code inside the { } runs once for each value in the sequence.

In every iteration of the loop, the value assigned to the variable is one of the values in the sequence.

This is important to understand as we can use this variable to perform similar tasks on different things.

 Practice: loops (reminder)

Write a code using loops that prints all numbers from 1 to 15.

Practice: loops (reminder)

Write a code using loops that prints all numbers from 1 to 15.

```
for(num in 1:15){
   print(num)
}
```

```
numbers_to_print = 1:15
for(num in numbers_to_print){
   print(num)
}
```

 Practice: loops 1

Write a code using loops that returns the sum of all numbers from 35 to 55 (without using the `sum()` function).

 Practice: loops 1

Write a code using loops that returns the sum of all numbers from 35 to 55 (without using the `sum()` function).

```
numbers = 35:55
sum = 0

for(num in numbers){
    sum = sum + num
}
print(sum)
```

Practice: loops 2

Write a code using loops and conditions that reports numbers bigger than 50 in this vector:

```
vec = c(41, 55, 2, 89, 95, 50)
```

Practice: loops 2

Write a code using loops and conditions that reports numbers bigger than 50 in this vector:

```
vec = c(41, 55, 2, 89, 95, 50)
```

```
for(number in vec){

   if(number > 50){
      print(number)}
}
```

Practice: loops 2: alternate answer

Write a code using loops and conditions that reports numbers bigger than 50 in this vector:

```
vec = c(41, 55, 2, 89, 95, 50)
```

Length of the vector given, This means the sequence is 1,2,3,4,5,6 which are the index numbers for the vector.

`vec[i]` is the i'th element of the vector

```
for(i in 1:6){

   if(vec[i] > 50){
      print(vec[i])}
}
```

Practice: loops 3

Write a code using loops and conditions that sums numbers bigger than 50 in this vector:

```
vec = c(41, 55, 2, 89, 95, 50)
```

 Practice: loops 3: TABLE OF VARS

Write a code using loops and conditions that sums numbers bigger than 50 in this vector:

```
vec = c(41, 55, 2, 89, 95, 50)
```

```
sum = 0
for(number in vec){

    if(number > 50){
        sum = sum + number
        }
}
print(sum)
```

	Loop variable (number)	Is condition satisfied	sum
Before loop starts	NA	NA	0
Iteration 1			
Iteration 2			
Iteration 3			
Iteration 4			
Iteration 5			
Iteration 6			

 Practice: loops 3: TABLE OF VARS

Write a code using loops and conditions that sums numbers bigger than 50 in this vector:

```
vec = c(41, 55, 2, 89, 95, 50)
```

```
sum = 0
for(number in vec){

    if(number > 50){
        sum = sum + number
        }
}
print(sum)
```

	Loop variable (number)	Is condition satisfied	sum
Before loop starts	NA	NA	0
Iteration 1	41	FALSE	0
Iteration 2			
Iteration 3			
Iteration 4			
Iteration 5			
Iteration 6			

 Practice: loops 3: TABLE OF VARS

Write a code using loops and conditions that sums numbers bigger than 50 in this vector:

```
vec = c(41, 55, 2, 89, 95, 50)
```

```
sum = 0
for(number in vec){

    if(number > 50){
        sum = sum + number
        }
}
print(sum)
```

	Loop variable (number)	Is condition satisfied	sum
Before loop starts	NA	NA	0
Iteration 1	41	FALSE	0
Iteration 2	55	TRUE	55
Iteration 3			
Iteration 4			
Iteration 5			
Iteration 6			

 Practice: loops 3: TABLE OF VARS

Write a code using loops and conditions that sums numbers bigger than 50 in this vector:

```
vec = c(41, 55, 2, 89, 95, 50)
```

```
sum = 0
for(number in vec){

    if(number > 50){
        sum = sum + number
        }
}
print(sum)
```

	Loop variable (number)	Is condition satisfied	sum
Before loop starts	NA	NA	0
Iteration 1	41	FALSE	0
Iteration 2	55	TRUE	55
Iteration 3	2	FALSE	55
Iteration 4	89	TRUE	55 + 89
Iteration 5	95	TRUE	55 + 89 + 95
Iteration 6	50	FALSE	55 + 89 + 95

Practice: loops 4

Write a code using loops and conditions that finds the largest number in a vector. Do not use the function `max()`.

```
vec = c(41, 55, 2, 89, 95, 50)
```

Practice: loops 4

Write a code using loops and conditions that finds the largest number in a vector. Do not use the function `max()`.

```
vec = c(41, 55, 2, 89, 95, 50)
```

```
largest_num = vec[1]

for(num in vec[2:6]){
    if(num > largest_num){
        largest_num = num
        }
}
print(largest_num)
```

Practice: loops 4: write it with looping over indexes

Write a code using loops and conditions that finds the largest number in a vector. Do not use the function `max()`.

```
vec = c(41, 55, 2, 89, 95, 50)
```

```
largest_num = vec[1]

for(i in 2:6){
    if(vec[i] > largest_num){
        largest_num = vec[i]
        }
}
print(largest_num)
```

Practice: loops 5

Write a code using loops that prints the contents of this vector in reverse:

```
vec = c("n", "e", "m", "o", "w")
```

Hint: this is one of those cases where using the index in loop is helpful.

Practice: loops 5

Write a code using loops that prints the contents of this vector in reverse:

```
vec = c("n", "e", "m", "o", "w", "x")
```

Hint: this is one of those cases where using the index in loop is helpful.

```
for(i in 6:1){
    print(vec[i])
}
```

Some tips for the next exercise:

- You can use the function `c()` to add to a vector: `t = c(t, 2)` will add 2 at the end of whatever vector `t` was.
  - Remember how we used c(1, 2, 3, 4) do concatenate numbers into a vector.
  - You can also:

```
t = c(1, 2, 3)
t2 = c(t, t)
t3 = c(t, 4, 5, 6)
```

- You can use the function `vector()` to define an empty vector to fill it in later.

```
t = vector()
t = c(t, 2)
t = c(t, 3)
```

Try it yourself. What will t be?

Practice: loops 6

Write a code using loops that reverts this vector and creates another vector called `reverse_vec`.

```
vec = c("n", "e", "m", "o", "w", "x")
```

Practice: loops 6

Write a code using loops that reverts this vector and creates another vector called `reverse_vec`.

```
vec = c("n", "e", "m", "o", "w", "x")

  #create a new empty vector
  reverse_vec = vector()

  #fill the new vector with reversed order
  for(i in 6:1){
      reverse_vec = c(reverse_vec, vec[i])
  }

  print(reverse_vec)
```

General tip for defining loops

- Try to define the sequence in your loop as dynamically as possible. For instance in the loop:

vec = c(“n”, “e”, “m”, “o”, “w”, “x”)

```
for(i in 6:1){
    reverse_vec = c(reverse_vec, vec[i])
}
```

- Instead of 6 (which is specific to this one vector) use `length(vec)`
- This way if your vector changes to a shorter or longer one, your code still works.

Break

Data manipulation in R: basics

In R, the **working directory** is the folder on your computer where R reads and saves files by **default**.
(If no path is specified explicitly)

You can check your current working directory using: `getwd()`

```
> getwd()
[1] "/Users/sana/Documents"
```

You can change your working directory using:

```
setwd("path/to/your/desired/directory")
```

This function takes a "string" as an input. This string must have the format of a path.

Data manipulation in R: basics

- In R, the **working directory** is the folder on your computer where R reads and saves files by **default**.
  (If no path is specified explicitly)

- You can check your current working directory using: `getwd()`

```
> getwd()
[1] "/Users/sana/Documents"
```

- You can change your working directory using:

```
setwd("path/to/your/desired/directory")
```

This function takes a "string" as an input. This string must have the format of a path.

- Create a folder on your computer, and call it "R_workshop"

- Set your working directory to this folder you created, and check using `getwd()`

```
setwd("/users/sana/Documents/R_workshop/")
```

Data manipulation in R: delimitation

- When working with tables and data frames, there are different formats in which you can read and write them.
- Usually, each row is written on a separate line, and columns are separated using “delimiters”.
- Some of the most commonly used delimiters are commas, tabs, and colons.
  - **CSV** file extensions are comma separated

```
ID,Name,Age
1,John,25
2,Sara,30
```

  - **TSV** file extensions are tab-separated

```
ID  Name    Age
1   John    25
2   Sara    30
```

Data manipulation in R: load data into R

```
setwd("path/to/your/file/directory")

dataset = read.delim(file = "employees.tsv", sep = "\t")

dataset = read.delim(file = "employees.csv", sep = ",")

read.csv()
```

Data manipulation in R: load data into R

```
setwd("path/to/your/file/directory")

dataset = read.delim(file = "employees.tsv", sep = "\t")

dataset = read.delim(file = "employees.csv", sep = ",")

read.csv()
```

"\t"	This string is understood as a space (like pressing on the tab button)
"\n"	This string is understood as a new line (like pressing enter)

Practice: loading data basics

- Set the working directory to where your download files are. Load the “employees” datasets using the `read.delim()` function and store it in a variable called `dataset`.

```
setwd(“path/to/your/file/directory”)
dataset = read.delim(file = “employees.tsv”, sep = “\t”)
```

- Use the `head()` function to show the first 5 rows of the dataset.

```
head(dataset)
```

- Use the `colnames()` function to see the column names of the dataset.

```
colnames(dataset)
```

If you type `dataset$` and press the ‘tab’ button while your cursor is after the `'$'`, RStudio will suggest the names of columns for you.
(This also works for functions and previously defined variables)

Practice: loading data basics

- Use the `dim()` function to check the dimensions of the dataset. You can also use `nrow()` and `ncol()`.

```
dim(dataset)
nrow(dataset)
ncol(dataset)
```

- Store the first 50 rows in a data frame called `sub_data1` and the second 50 rows in another called `sub_data2`.

```
sub_data1 = dataset[1:50, ]
sub_data2 = dataset[51:100, ]
```

- The `rbind()` function can be used to bind multiple data frames or matrices that have the exact same number of columns. Use this function to bind the two sub_data frames you created earlier.

```
new_df = rbind(sub_data1, sub_data2, sub_data1)
```

Practice: loading data basics

- Add a column called ‘ID’ and let each employees ID be the row number (so just consecutive numbers from 1 to 100).

```
dataset$ID = 1:100
dataset[, “ID”] = 1:100
```

- Print the `Department` column. Then get the unique values represented in this column using the `unique()` function.

```
print(dataset$Department)
unique(dataset$Department)
```

- Find the minimum, maximum, and mean `Performance_Score` using `min(), max(), mean()` functions.

```
min(dataset$Performance_Score)
max(dataset$Performance_Score)
mean(dataset$Performance_Score)
```

Practice

- Go through every employee listed in this dataset. If their performance score is less than 2.5, add their name to a vector called `vec`.

  (HINT: remember you can create empty vectors using `vector()` and add to them using the `c()` function)

Practice

Go through every employee listed in this dataset. If their performance score is less than 2.5, add their name to a vector called `vec`.

(HINT: remember you can create empty vectors using `vector()` and add to them using the `c()` function)

```
#create an empty vector
vec= vector()

for(i in 1:nrow(dataset)){

    if(dataset$Performance_score[i] < 2.5){
        vec = c(vec, dataset$Employee_Name[i])
        }
}

Print(vec)
```

Practice

- Go through every employee listed in this dataset. If they are remote workers and their satisfaction level is higher than 75%, add their name to a vector called `vec`.

  (HINT: remember you can create empty vectors using `vector()` and add to them using the `c()` function)

Practice

Go through every employee listed in this dataset. If they are remote workers and their satisfaction level is higher than 75%, add their name to a vector called `vec`.

(HINT: remember you can create empty vectors using `vector()` and add to them using the `c()` function)

```
#create an empty vector
vec= vector()

for(i in 1:nrow(dataset)){

    if((dataset$Remote_Worker[i] == TRUE) && (dataset$Satisfaction_Level[i] > 75)){

        vec = c(vec, dataset$Employee_Name[i])

        }
}
print(vec)
```

Data manipulation in R: The `which()`

- This is a very useful function that returns the **position (index)** of positions in a vector for which a condition is true (hence the name ‘which’). In many cases does what you otherwise would need a loop for.

Data manipulation in R: The `which()`

This is a very useful function that returns the **position (index)** of positions in a vector for which a condition is true (hence the name 'which'). In many cases does what you otherwise would need a loop for.

```r
x <- c(10, 20, 30, 40, 50)

which(x > 25)
```

Output:

```
[1] 3 4 5
```

`which(`*`condition on a vector`*`)`

- which() gives positions (indices), not values.
- It's useful for finding row numbers in data frames.
- Combine it with [] to extract values: `x[which(x > 25)]`

Data manipulation in R: The `which()`

- What happens inside of which, is that it gives you the position of TRUE's in a vector of logical values.
- When you run `x > 25`, it returns a vector of `TRUE`'s and `FALSE`'s where each is the result of the comparison for each element of `x`:

```
x <- c(10, 20, 30, 40, 50)
```

```
x > 25
```

```
[1] FALSE FALSE  TRUE  TRUE  TRUE
```

- Now when you wrap `which()` around `x > 25`, it tells you where the position of `TRUE`'s is, in the comparison.
- This is important to know, that what goes into `which()` should be a vector of logicals.

```
which(c(TRUE, FALSE, TRUE, TRUE, FALSE, FALSE))
```

Practice: using `which()`

- Without using a loop, get the row numbers for employees for which their satisfaction is above 75.

Practice: using `which()`

- Without using a loop, get the row numbers for employees for which their satisfaction is above 75.

```
satisfied_employees_index = which(dataset$Satisfaction_Score > 75)
```

Practice: using `which()`

- Without using a loop, get the row numbers for employees for which their satisfaction is above 75.

```
satisfied_employees_index = which(dataset$Satisfaction_Score > 75)
```

- Without using a loop, get the names of employees for which their satisfaction is above 75.

Practice: using `which()`

- Without using a loop, get the row numbers for employees for which their satisfaction is above 75.

```
satisfied_employees_index = which(dataset$Satisfaction_Score > 75)
```

- Without using a loop, get the names of employees for which their satisfaction is above 75.

```
satisfied_names = dataset$Employee_Name[which(dataset$Satisfaction_Score > 75)]
```

```
satisfied_names = dataset[which(dataset$Satisfaction_Score > 75), “Employee_Name”]
```

Practice: using `which()`

- Using the `which()` function, create a new dataset called `remote_employees` and store the data for remote workers in it.

Practice: using `which()`

Using the `which()` function, create a new dataset called `remote_employees` and store the data for remote workers in it.

```
remote_employees = dataset[which(dataset$Remote_Worker), ]
```

```
remote_employees = dataset[which(dataset$Remote_Worker == TRUE), ]
```

Data manipulation in R: The `%in%` operator

- `%in%` checks if values from one set exist in another set. It returns TRUE or FALSE for every value from the left-hand side that is matched against the right-hand side.

```
set1 %in% set2
```

- For every value in `set1`, a logical value is returned. `TRUE` if it is found in `set2`, and `FALSE` if not.

```
5 %in% c(2, 3, 5, 7)
```

```
[1] TRUE
```

```
c(5, 10, 15) %in% c(2, 5, 10, 20)
```

```
[1]  TRUE  TRUE FALSE
```

Data manipulation in R: The `%in%` operator

`%in%` can be combined with `which()` to get the index of the elements in `set1` that are found in `set2`. (Recall that `which()` gives you the positions for which a condition is TRUE).

```
set1 %in% set2
```

```r
x <- c(10, 20, 30, 40, 50)

which(x %in% c(20, 40))
```

```
[1] 2 4
```

Data manipulation in R: writing tables in R

You can write tables you create in R using the `write.table()` function.

```
write.table(dataset,
            file = "path/to/directory/filename.tsv",
            sep = "\t",
            row.names=FALSE,
            col.names=TRUE,
            quote = TRUE
            )
```

1. This is the variable name for the table you want to save

2. This is the path to your file, with the file name at the end. If you don't specify a path, it will save in your working directory.

Data manipulation in R: writing tables in R

You can write tables you create in R using the `write.table()` function.

```
write.table(dataset,
            file = “path/to/directory/filename.tsv”,
            sep = “\t”,
            row.names=FALSE,
            col.names=TRUE,
            quote = TRUE
            )
```

3. This is the separator for columns. Usually it will be one of “\t” or “,” or “:”

Data manipulation in R: writing tables in R

You can write tables you create in R using the `write.table()` function.

```
write.table(dataset,
            file = “path/to/directory/filename.tsv”,
            sep = “\t”,
            row.names=FALSE,
            col.names=TRUE,
            quote = TRUE
            )
```

4. These two specify whether you want the column names and rownames to be saved as well. Usually it’s best to set column names to true and row names to false.

Data manipulation in R: writing tables in R

You can write tables you create in R using the `write.table()` function.

```
write.table(dataset,
            file = “path/to/directory/filename.tsv”,
            sep = “\t”,
            row.names=FALSE,
            col.names=TRUE,
            quote = TRUE
            )
```

5. This specifies whether you want to wrap every element in every “cell” of your table to be wrapped in quotation marks. This way if you load the table into another program, everything will be in a “string” type.

Practice: writing data in R

- Create a subset of the data, from individuals working only in Finance and Marketing. Write this data frame using the `write.table()` function.

Practice: writing data in R

Create a subset of the data, from individuals working only in Finance and Marketing. Write this data frame using the `write.table()` function.

```
targets = c("Finance", "Marketing")
subset = dataset[which(dataset$Department %in% targets), ]
write.table(subset,
            file = "/users/sana/Documents/subset.tsv",
            row.names = FALSE,
            quote = FALSE)
```

It is good practice to be consistent with how you write data.

Break

Data visualization: Base R

- In base R, you can use built-in functions to make simple plots without extra libraries.
- These functions are easy to use, but not very versatile.

Scatter plot in blue

```
x = c(10, 20, 30, 40, 50)
y = c(5, 15, 25, 35, 45)

plot(x, y, main = "Scatter Plot Example", xlab = "X Values", ylab = "Y Values", col = "blue", pch = 16)
```

Line plot in red

```
x = c(10, 20, 30, 40, 50)
y = c(5, 15, 25, 35, 45)

plot(x, y, type = "l", col = "red", lwd = 2, main = "Line Plot Example")
```

There are also simple commands to plot density plots and histograms.

Data visualization: ‘ggplot2’

- `ggplot2` is a powerful data visualization package in R that makes customizable plots.
- The idea behind plotting with `ggplot2` is that you start with a blank plot, and you add layers on top of it.
- This allows for a lot of flexibility in terms of overlaying different types of plots on each other, and also aesthetics.
- `ggplot2` works best when the data you want to visualize is in form of a **data frame**. This makes adding layers easier.

https://r-graphics.org/

Data visualization: ‘ggplot2’ general structure

- This is the bit that initializes the plot.
- This is also where you define the dataset you want to plot.
- If defined here, this is the dataset that will be used to plot the subsequent layers that you add (unless otherwise specified).

```
p = ggplot(data = dataset_name) +

    geom_point(...) +

    geom_line(...)
```

Data visualization: ‘ggplot2’ general structure

```
p = ggplot(data = dataset_name) +

    geom_point(...) +

    geom_line(...)
```

You add layers on top of the initial plot using “+”.

Data visualization: ‘ggplot2’ general structure

```
p = ggplot(data = dataset_name) +

    geom_point(...) +

    geom_line(...)
```

Depending on what kind of plot you want to add, you use each of the family of functions: `geom_*()`

- `geom_point()` is for scatterplots.
- `geom_line()` is for line plots.
- `geom_boxplot()` is for boxplots.
- ...

Data visualization: ‘ggplot2’ general structure

Initialization of the plot

```
p = ggplot(data = dataset_name) +

    geom_point(aes(x = c1, y = c2), color = “red”, size = 2, …)
```

Mapping: mandatory

Other aesthetics and options: optional

Data visualization: ‘ggplot2’ general structure

Initialization of the plot

```
p = ggplot(data = dataset_name) +

    geom_point(aes(x = c1, y = c2), color = “red”, size = 2, …)
```

Mapping: mandatory

Other aesthetics and options: optional

- The `aes()` argument is mandatory for `ggplot`. This is where you determine the **mapping** of your plot, ie, what columns of your dataset determine which aspects of your plot (x-axis, y-axis, colour mapping, size mapping etc.)
- You **must determine the `x` and `y` aspects of `aes()` at the minimum**. There are other optional arguments that can be defined, but **x and y are mandatory**.

Data visualization: ‘ggplot2’ general structure

Initialization of the plot

```
p = ggplot(data = dataset_name) +

    geom_point(aes(x = c1, y = c2), color = “red”, size = 2, …)
```

Mapping: mandatory

Other aesthetics and options: optional

- The `aes()` argument is mandatory for `ggplot`. This is where you determine the **mapping** of your plot, ie, what columns of your dataset determine which aspects of your plot (x-axis, y-axis, colour mapping, size mapping etc.)
- You **must determine the `x` and `y` aspects of `aes()` at the minimum**. There are other optional arguments that can be defined, but **x and y are mandatory**.
- The syntax is:

  `aes(x = c1, y = c2)`

  where `c1` and `c2` are the **names** of the columns in your data frame which you want to be plotted on the x and y axis of your graph.

Practice ‘ggplot2’: load data

Load the other dataset provided to you called `'sales.tsv'` and call it `sales`. Run the `head()` function on the dataset to take a look at the first 5 rows.

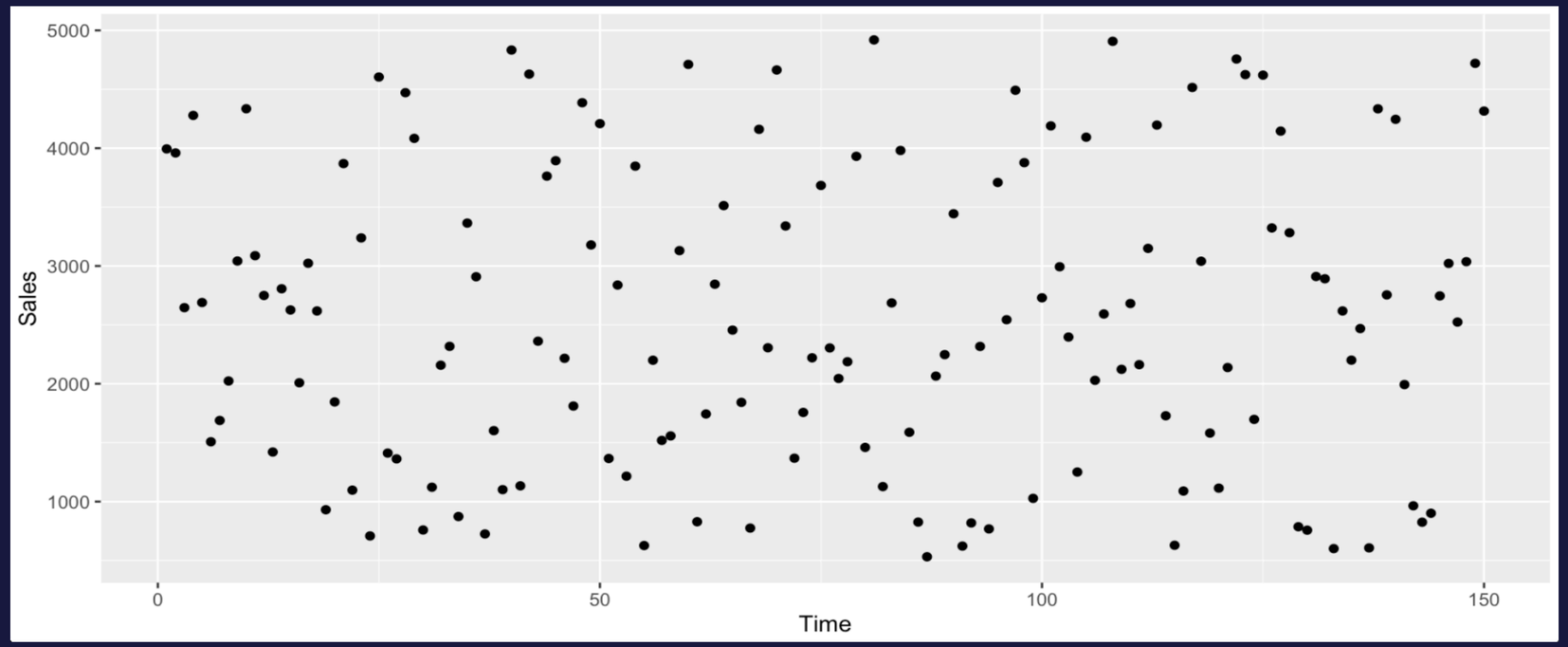
```
sales = read.delim("/users/sana/Documents/sales.tsv")
head(sales)
```

```
> head(sales)
        Date Category   Sales Profit_Margin Customer_Rating Online_Purchase Time
1 2023-01-01        A 3994.00          66.4              2.6            TRUE    1
2 2023-01-02        B 3959.84          79.8              2.3           FALSE    2
3 2023-01-03        C 2646.14          94.1              1.0            TRUE    3
4 2023-01-04        A 4278.34          80.6              3.2           FALSE    4
5 2023-01-05        B 2689.89           2.8              1.2            TRUE    5
6 2023-01-06        A 1507.78          59.7              3.3           FALSE    6
```

Practice ‘ggplot2’: basic scatter plot

Let’s plot a simple scatter plot where `Time` is on the x-axis and `Sales` is on the y-axis.

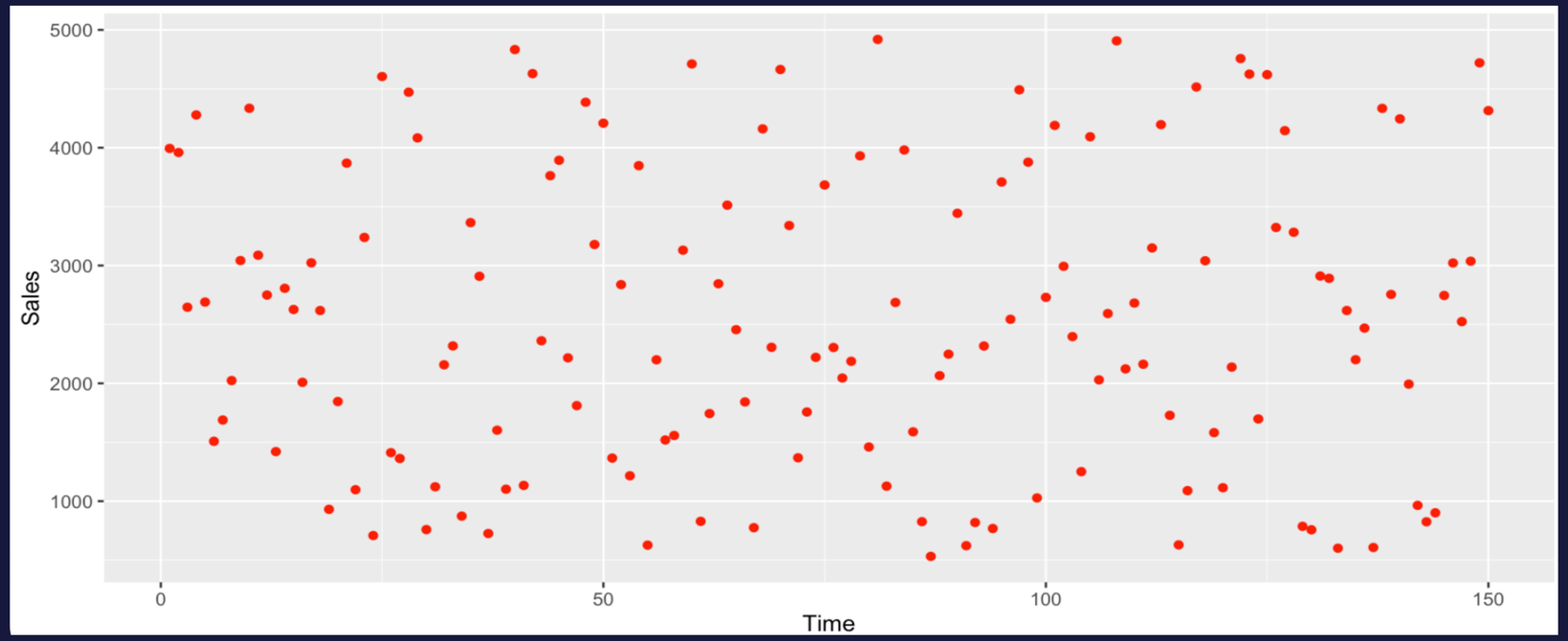
```
ggplot(data = sales) +
   geom_point(aes(x = Time, y = Sales))
```



Practice ‘ggplot2’: coloured scatterplot

Let’s plot a simple scatter plot where `Time` is on the x-axis and `Sales` is on the y-axis, and make it red.

```
ggplot(data = sales) +
   geom_point(aes(x = Time, y = Sales), color = “red”)
```



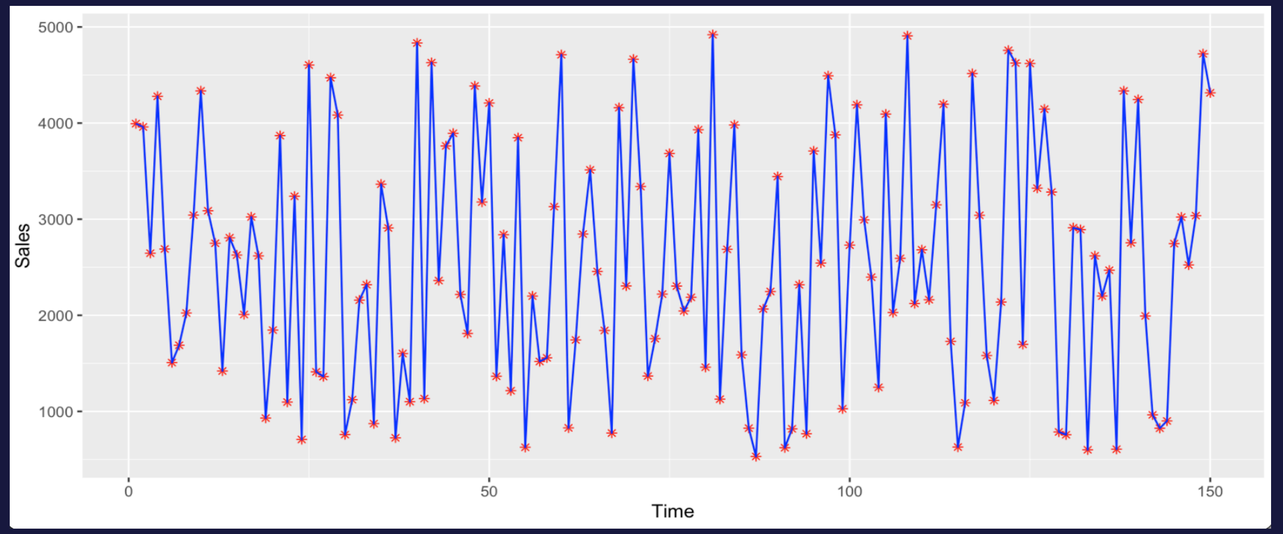
Practice ‘ggplot2’: add a new layer

Now let’s add a layer to the previous plot, where we connect the points with lines. We also add colour to the plot.

We make the points red and the lines blue. We also change the shape for the points to asterisks (code #8).

```
ggplot(data = sales) +
   geom_point(aes(x = Time, y = Sales), color = “red”, shape = 8)+

   geom_line(aes(x = Time, y = Sales), color = “blue”)
```



Final notes.

- Learning programming and getting used to it, takes time. Allow yourself to trial and error, and do not be afraid to try and run anything.

- Do not copy and paste (at least in the initial stages of your learning journey)! The process of typing code prompts you to think about what you are typing, and helps you understand. This helps you learn!

- Almost any programming related information can be found online these days! Don’t be shy to google your questions.

Future learning directions.

- If you find the time or bandwidth for self-learning, try learning about writing **functions and variable scopes.**
- After learning about functions, a very useful family of functions to learn about is are the ‘`apply`’ family. These are a family of functions that increase speed and efficiency.
- The ‘`stringr`’ package in R is a very useful package to learn to work with.
- Don’t get too caught up in fancy, intricate packages. Most of what you need can already be done efficiently in base R.
- A useful and widely used package for data wrangling in R is ‘`tidyverse`’. You can do everything ‘`tidyverse`’ does in base R. This is a matter of personal choice.
- Good luck and happy programming! 

Things we did not have time for:

Practice: the `%in%` operator

- Using the `%in%` operator and the `which()` function, return the name of employees that are either in “IT” or “HR”.

Practice: the `%in%` operator

- Using the `%in%` operator and the `which()` function, return the name of employees that are either in “IT” or “HR”.

```
targets = c(“HR”, “IT")
indices = which(dataset$Department %in% target)
names = dataset$Employee_name[indices]
print(names)
```

- OR:

```
targets = c(“HR”, “IT")
names = dataset$Employee_name[which(dataset$Department %in% target)]
print(names)
```

- As practice, try to write this without the %in% operator and using which() only.

 Practice: the `%in%` operator

- Using the `%in%` operator check if “Hannah Smith” and “Laura Camelo” are employees in this company.

 Practice: the `%in%` operator

Using the `%in%` operator check if “Hannah Smith” and “Laura Camelo” are employees in this company.

```
names_to_check = c(“Hannah Smith”, “Laura Camelo")
indices = which(names_to_check %in% dataset$Employee_name)
```

 Watch out for what you want to check: the order of set1 and set 2 matters!

(What would happen if you reversed the order in this example? What about the previous exercise?)

Practice ‘ggplot2’: try it on your own!

- Plot a scatter plot where `Time` is on the x-axis and `Customer_Rating` is on the y-axis. Make the points orange.

  Then connect the points using lines.

Practice ‘ggplot2’: try it on your own!

Plot a scatter plot where `Time` is on the x-axis and `Customer_Rating` is on the y-axis. Make the points orange.

Then connect the points using lines.

```
ggplot(data = sales) +
   geom_point(aes(x = Time, y = Customer_Rating), color = “orange”)+

   geom_line(aes(x = Time, y = Customer_Rating), color = “orange”)
```

Data visualization: ‘ggplot2’ mapping aesthetics

- In base R, you can use built-in functions to make simple plots without extra libraries.
- These functions are easy to use, but not very versatile.

```
p = ggplot(data = dataset_name) +

    geom_point(aes(x = c1, y = c2, color = c3), size = 2, …)
```

- If we put `color` inside the `aes()`, then it becomes a mapping: **that layer will no longer be plotted in a fixed color.**
- Now, colour will be mapped to the contents of c3 (some column in the data frame). Meaning that the colour for each data point is determined by its value at column c3.
- You can do this for all aesthetics: **color, size, shape, etc.**

Practice ‘ggplot2’: colour mapping

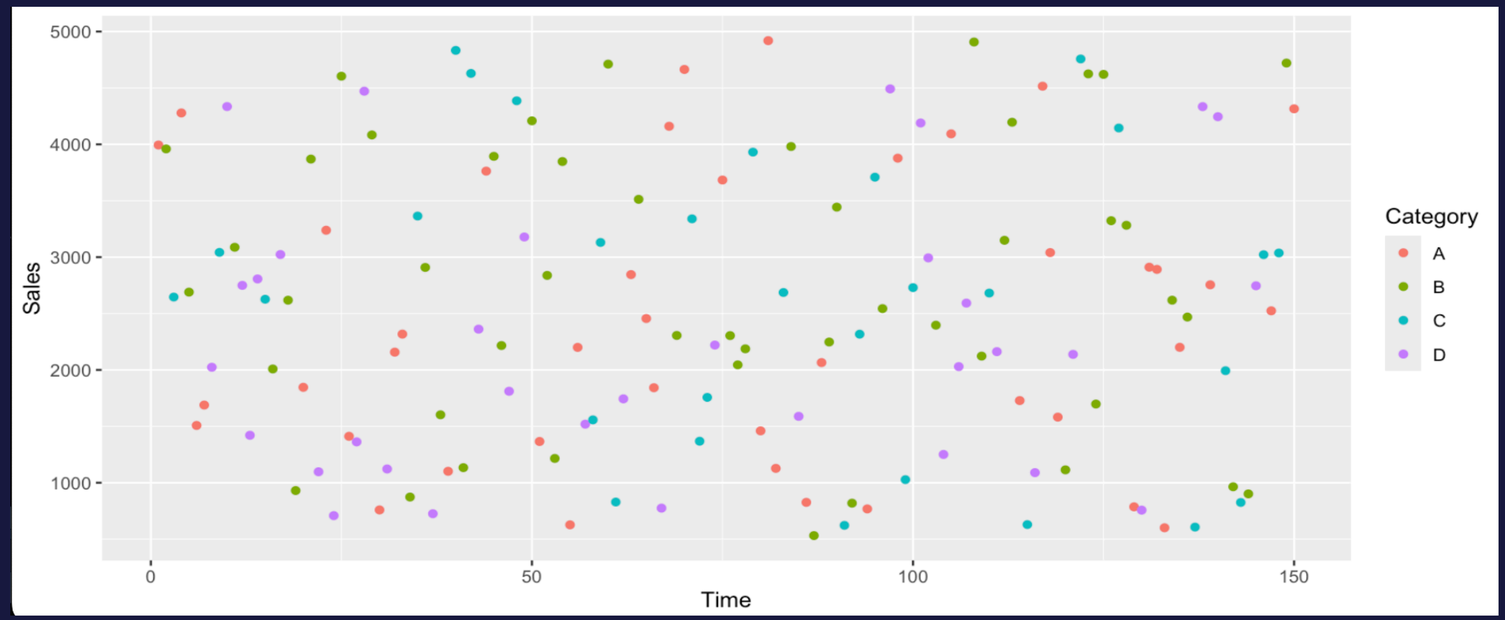
Make a scatter plot of `Sales` over `Time`. Map the colour of the scatter plot to the ‘Category’ column such that each category will have a different colour.

```
ggplot(data = sales) +
   geom_point(aes(x = Time, y = Sales, color = Category))
```

Practice ‘ggplot2’: colour mapping

Make a scatter plot of `Sales` over `Time`. Map the colour of the scatter plot to the ‘Category’ column such that each category will have a different colour.

```
ggplot(data = sales) +
  geom_point(aes(x = Time, y = Sales, color = Category))
```



Practice ‘ggplot2’: shape mapping

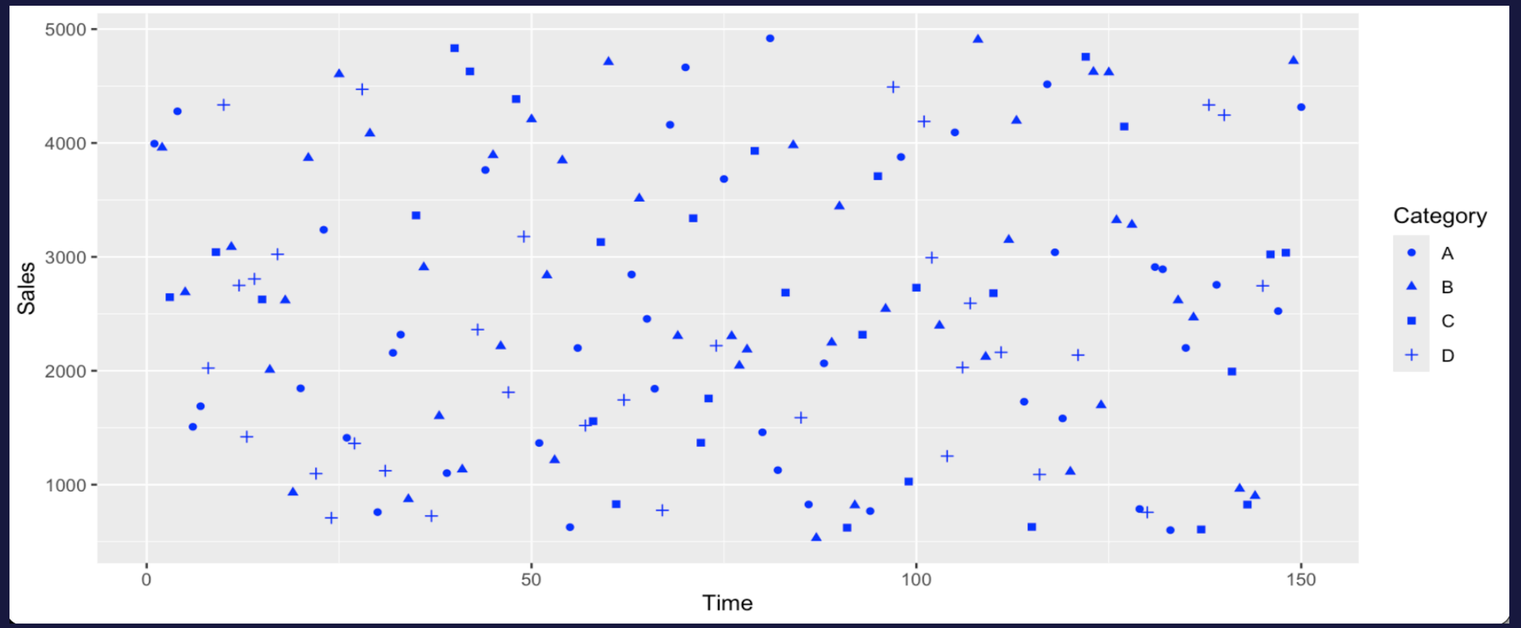
- Make a scatter plot of `Sales` over `Time`. Map the **shape** of the scatter plot to the ‘Category’ column such that each category will have a different shape.

```
ggplot(data = sales) +
   geom_point(aes(x = Time, y = Sales, shape = Category),
color = “blue”)
```

Practice ‘ggplot2’: shape mapping

Make a scatter plot of `Sales` over `Time`. Map the **shape** of the scatter plot to the ‘Category’ column such that each category will have a different shape.

```
ggplot(data = sales) +
   geom_point(aes(x = Time, y = Sales, shape = Category),
color = “blue”)
```



Practice ‘ggplot2’: shape and color mapping

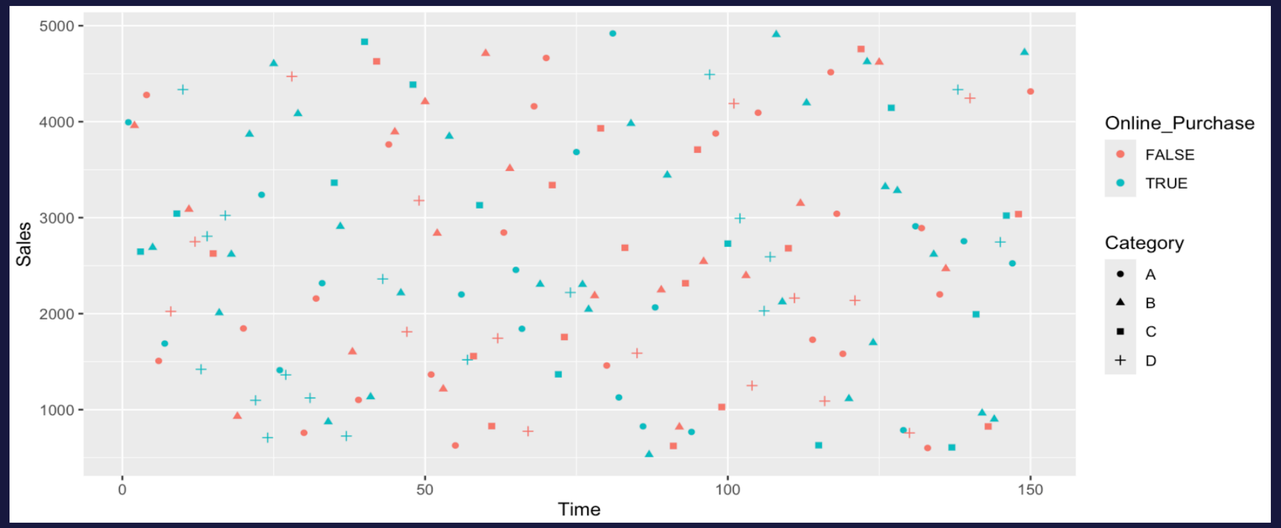
- Make a scatter plot of `Sales` over `Time`. Map the **shape** of the scatter plot to the ‘Category’ column such that each category will have a different shape. Map the colour of the points to the ‘Online_Purchase’ so online and offline purchases have different colours.

```
ggplot(data = sales) +
   geom_point(aes(x = Time, y = Sales, shape = Category,
color = Online_Purchase))
```

Practice ‘ggplot2’: shape and color mapping

- Make a scatter plot of `Sales` over `Time`. Map the **shape** of the scatter plot to the ‘Category’ column such that each category will have a different shape. Map the colour of the points to the ‘Online_Purchase’ so online and offline purchases have different colours.

```
ggplot(data = sales) +
   geom_point(aes(x = Time, y = Sales, shape = Category,
color = Online_Purchase))
```



Practice ‘ggplot2’: colour mapping and custom colours

- Make a scatter plot of `Sales` over `Time`. Map the **shape** of the scatter plot to the ‘Category’ column such that each category will have a different colour.

```
ggplot(data = sales) +

  geom_point(aes(x = Time, y = Sales, color = Category))+

  scale_color_manual(values = c("black", "darkblue", "blue", “lightblue"))
```

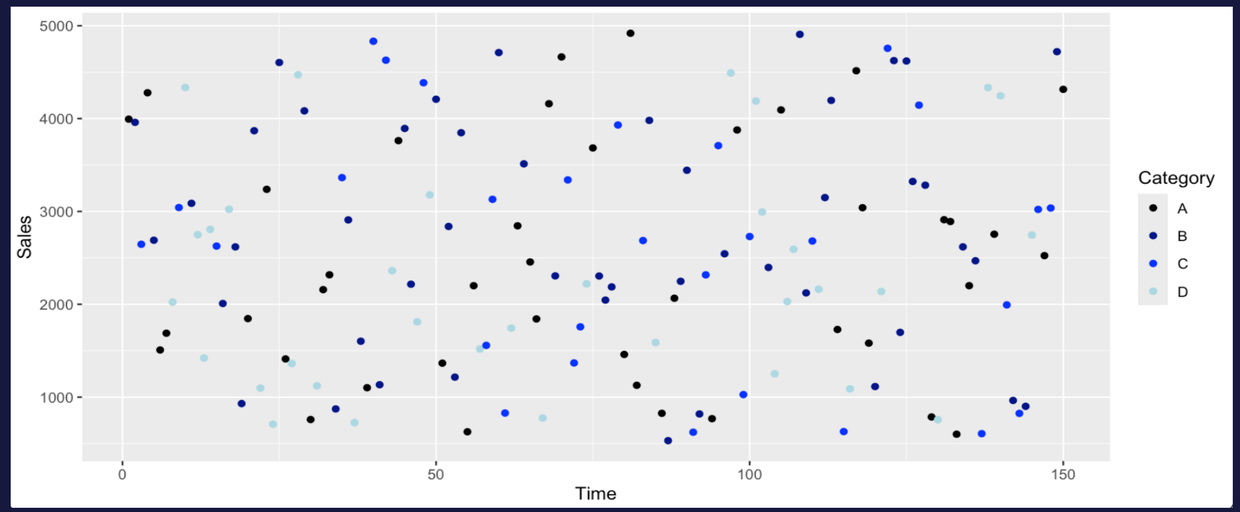
Practice ‘ggplot2’: colour mapping and custom colours

Make a scatter plot of `Sales` over `Time`. Map the **shape** of the scatter plot to the ‘Category’ column such that each category will have a different colour.

```
ggplot(data = sales) +

  geom_point(aes(x = Time, y = Sales, color = Category))+

  scale_color_manual(values = c("black", "darkblue", "blue", “lightblue"))
```



Data visualization: scale manuals

```
scale_colour_manual()

scale_fill_manual()

scale_size_manual()

scale_shape_manual()

scale_linetype_manual()

scale_linewidth_manual()

scale_alpha_manual()
```

Data visualization: saving ggplots

```
p = ggplot(data = sales) + geom_point(aes(x = Time, y = Sales))

ggsave(p,
       file = “/users/sana/Documents/plot.pdf”,
       device = “pdf”,
       height = 10,
       width=10)

ggsave(p,
       file = “/users/sana/Documents/plot.png”,
       device = “png”,
       height = 10,
       width=10)

print(p)

p + geom_line(…)
```

Data visualization: other features

```
p = ggplot(data = sales) + geom_point(aes(x = Time, y = Sales))+


xlab("label you want on x axis")+
ylab("label you want on y axis")+
ggtitle("title you want to appear at the top of the plot")+
ylim(c(0, 100))+ #limits you want on y-axis
xlim(c(0, 150))+ #limits you want on x-axis
theme(…)
```

https://r-graphics.org/RECIPE-APPEARANCE-TEXT-APPEARANCE.html