FUNCTIONAL PROGRAMMING

Gerko Vink

g.vink@uu.nl

Methodology & Statistics @ Utrecht University

9 Jun 2025

DISCLAIMER

I owe a debt of gratitude to many people as the thoughts and code in these slides are the process of years-long development cycles and discussions with my team, friends, colleagues and peers. When someone has contributed to the content of the slides, I have credited their authorship.

Images are either directly linked, or generated with StableDiffusion or DALL-E. That said, there is no information in this presentation that exceeds legal use of copyright materials in academic settings, or that should not be part of the public domain.

Warning

You **may use** any and all content in this presentation - including my name - and submit it as input to generative AI tools, with the following **exception**:

- You must ensure that the content is not used for further training of the model

SLIDE MATERIALS AND SOURCE CODE

Materials

- lecture slides on Moodle
- course page: www.gerkovink.com/sur
- source: github.com/gerkovink/sur

RECAP

Gisteren hebben we deze onderwerpen behandeld:

- Beschrijvende statistiek
- Kruistabellen en frequentieverdelingen
- χ^2-toets
- Andere toets- en associatiematen
- Simpele lineaire regressie
- Analyses draaien op groepen

TODAY

Vandaag behandelen we de volgende onderwerpen:

- Zelf functies ontwikkelen, gebruiken en debuggen
- Map / Reduce workflows
- Binaire operators
- Trekken uit verdelingen
- Random number generation

PACKAGES WE USE

```
library(dplyr)    # data manipulation
library(purrr)    # functional programming
library(furrr)    # parallel processing
library(magrittr) # flexible pipes
library(mice)     # for the boys data

# fix the random seed
set.seed(123)
```

WRITING FUNCTIONS

WRITE YOUR OWN FUNCTION

The function is `function()`

```
my_function <- function(arguments) {

  expressions

  return(output)

}
```

- `arguments` are input of the function
- `expressions` are operations performed on the arguments
- `output` an object containing the output (e.g. vector, matrix, list, etc.)
- `return` explicit return of the output (optional, but recommended!)

```
my_function <- function(arguments) {

  expressions

  output # less clear that this is returned

}
```

TOSSING A DIE

A function without arguments that simulates tossing a die

```
die <- function() {
  # throw die
  eyes <- sample(1:6, size = 1)
  # return the outcome
  return(eyes)
}
```

```
c(die(), die(), die())
```

```
[1] 3 6 3
```

DEFINING AN ARGUMENT

The argument n specifies the number of throws of the die

```
dice <- function(n) {
  # n is the number of dice to toss
  # replace = TRUE allows for repeated outcomes
  # returns a vector of length n
  return(sample(1:6, size = n, replace = TRUE))
}

dice(10)
```

```
[1] 2 2 6 3 5 4 6 6 1 2
```

MULTIPLE RETURNS

A function `dice(n)` returning a list with

- the outcomes of the `n` throws, their frequencies and their mean

```r
dice <- function(n) {
  # throw dice n times
  eyes <- sample(1:6, size = n, replace = TRUE)
  # prepare structured output
  return(list(outcomes = eyes,
              freqs    = table(eyes),
              mean     = mean(eyes)))
}
dice(10)
```

```
$outcomes
 [1] 3 5 3 3 1 4 1 1 5 3

$freqs
eyes
1 3 4 5
3 4 1 2

$mean
[1] 2.9
```

DEFAULT ARGUMENTS

The default is a fair die (each outcome has probability 1/6)

- the user can change this if so desired

```r
dice <- function(n, p = rep(1/6, 6)) {
  # throw dice n times with probability p
  eyes <- sample(1:6, size = n, replace = TRUE, prob = p)
  # prepare structured output
  return(list(outcomes  = eyes,
              frequency = table(eyes),
              mean      = mean(eyes)))
}
dice(100)
```

```
$outcomes
  [1] 5 5 3 2 1 1 6 6 2 4 6 3 3 3 2 4 4 4 2 2 3 4 3 1 2 4 6 2 5 3 2 6 1 4 5 2 4
 [38] 3 6 4 6 6 6 4 6 5 6 2 4 3 4 5 4 2 3 6 4 6 2 4 1 1 1 3 2 5 4 5 3 3 6 2 4 5
 [75] 5 3 4 1 4 1 1 5 4 2 1 3 2 1 6 2 5 1 5 4 5 3 3 3 4 1

$frequency
eyes
 1  2  3  4  5  6 
14 17 18 22 14 15 

$mean
[1] 3.5
```

UNFAIR DICE

The following command throws 100 unfair dice

- probabilities for rolling a 1, 2, 3, 4, 5 is 0.1
- probability for rolling a 6 is 0.5

```
c(rep(.1, 5), .5)
```

```
[1] 0.1 0.1 0.1 0.1 0.1 0.5
```

```
dice(100,  p = c(rep(.1, 5), .5))
```

```
$outcomes
  [1] 6 6 6 4 4 2 4 5 3 4 2 5 1 6 6 6 6 6 2 6 6 6 6 5 2 6 6 6 6 6 3 6 6 6 3 6 4
 [38] 6 6 3 5 6 6 6 4 6 2 5 4 4 6 3 2 3 2 6 5 6 3 6 6 3 1 1 6 6 1 4 1 6 6 4 6 3
 [75] 6 1 4 3 6 2 6 6 6 6 6 6 6 4 6 5 6 6 2 1 6 1 5 4 6 6

$frequency
eyes
 1  2  3  4  5  6 
 8  9 10 13  8 52 

$mean
[1] 4.6
```

APPLYING YOUR FUNCTION

apply()

The `apply()` function is used to apply a function to the rows or columns of a matrix or array. It takes three main arguments: the data, the margin (1 for rows, 2 for columns), and the function to apply.

```
calc_mean <- function(x) {
  return(mean(x, na.rm = TRUE))
}
# select random 10 rows from the numeric columns of boys
numboys <- boys %>% select(where(is.numeric))
which_rows <- sample(1:nrow(numboys), 10)
numboys <- numboys[which_rows, ]
# over the columns
apply(numboys, FUN = calc_mean, MARGIN = 2)
```

```
    age      hgt      wgt      bmi       hc       tv 
15.0746 168.3200  62.7300  21.2590  55.1300  10.5000 
```

```
# over the rows
apply(numboys, FUN = calc_mean, MARGIN = 1)
```

```
    5975     7062     6131     5897     4505     7073     7088     4487 
58.03867 67.37580 73.66980 68.45680 52.04250 62.35783 71.46680 48.28217 
    6693     2423 
72.43360 35.15900 
```

`lapply()`

`lapply()` does the same as `apply()`, but it is used for lists. It applies a function to each element of a list and returns a list of results.

```
lapply(numboys, FUN = calc_mean)
```

```
$age
[1] 15.0746

$hgt
[1] 168.32

$wgt
[1] 62.73

$bmi
[1] 21.259

$hc
[1] 55.13

$tv
[1] 10.5
```

`sapply()`

`sapply()` does the same as `lapply()`, but it simplifies the output to a vector or matrix if possible. It is useful when you want to avoid dealing with lists.

```
sapply(numboys, FUN = calc_mean)
```

```
     age      hgt      wgt      bmi       hc       tv 
 15.0746 168.3200  62.7300  21.2590  55.1300  10.5000 
```

tapply()

`tapply()` is used to apply a function to subsets of a vector, grouped by one or more factors. It is particularly useful for summarizing data based on grouping variables.

```
tapply(boys$hgt, boys$reg, FUN = calc_mean)
```

```
   north     east     west    south     city 
151.6316 133.9648 130.2783 128.0022 125.8577 
```

```
boys %>%
  group_by(reg) %>%
  summarise(mean_hgt = mean(hgt, na.rm = TRUE))
```

```
# A tibble: 6 × 2
  reg   mean_hgt
  <fct>    <dbl>
1 north    152. 
2 east     134. 
3 west     130. 
4 south    128. 
5 city     126. 
6 <NA>      73.0
```

MAP / REDUCE

`map()`

The `map()` function is part of the `purrr` package, which is designed for functional programming in R. It allows you to apply a function to each element of a list or vector, returning a list of results.

```
boys %>%
  split(.$reg) %>% # split the data by region
  map(~ lm(hgt ~ age, data = .x) %>% # map the linear model function
        coef()) # extract coefficients
```

```
$north
(Intercept)         age
  74.104664    6.376882

$east
(Intercept)         age
  73.229714    6.507535

$west
(Intercept)         age
  69.446550    6.646496

$south
(Intercept)         age
  70.410123    6.566541

$city
(Intercept)         age
  69.010565    6.724608
```

split()

```r
boys %>%
  split(.$reg)
```

```
$north
       age  hgt    wgt   bmi   hc  gen  phb tv   reg
127  0.093 56.0  5.410 17.25 40.0 <NA> <NA> NA north
198  0.117 57.0  5.260 16.18 40.0 <NA> <NA> NA north
238  0.142 58.0  5.220 15.51 40.1 <NA> <NA> NA north
248  0.147 57.3  4.950 15.07 36.8 <NA> <NA> NA north
873  0.594 70.8  8.970 17.89 45.2 <NA> <NA> NA north
911  0.673 71.0  9.000 17.85 46.5 <NA> <NA> NA north
1212 0.996 77.1 10.390 17.47 47.1 <NA> <NA> NA north
1278 1.040 77.5  9.300 15.48 46.3 <NA> <NA> NA north
1511 1.292 79.0 10.700 17.14 47.3 <NA> <NA> NA north
1617 1.481   NA 12.040    NA 47.5 <NA> <NA> NA north
1684 1.530 80.0 10.785 16.85 46.1 <NA> <NA> NA north
1877 1.793 86.0 13.400 18.11 47.0 <NA> <NA> NA north
1882 1.798 81.8 10.535 15.74 46.7 <NA> <NA> NA north
1927 1.848   NA 13.200    NA 50.0 <NA> <NA> NA north
2044 2.020 88.3 13.000 16.67 50.0 <NA> <NA> NA north
2168 2.198 94.2 14.980 16.88 51.0 <NA> <NA> NA north
2313 2.576 86.0 10.700 14.46 49.5 <NA> <NA> NA north
```

map()

The `map()` function is particularly useful for iterating over lists or vectors and applying a function to each element. It can be used to perform operations like calculations, transformations, or data extraction.

```
out <- boys %>%
  split(.$reg) %>% # split the data by region
  map(~ lm(hgt ~ age, data = .x) %>% # map the linear model function
        coef()) # extract coefficients

is.list(out)
```

```
[1] TRUE
```

```
names(out)
```

```
[1] "north" "east"  "west"  "south" "city"
```

```
out$city
```

```
(Intercept)         age 
  69.010565    6.724608 
```

map() ON LARGE LISTS

In the below example, we take a bootstrap sample (with replacement) from the boys data 1000 times, and then run a simple linear model on all 1000 bootstrap samples seperately.

```
sample_rows <- function(x) {
  out <- x[sample(1:nrow(x), replace = TRUE), ]
  return(out)
}
samples <- replicate(n = 1000, expr = sample_rows(boys), simplify = FALSE)
samples_lm <-
  samples %>%
  map(~.x %$%
        lm(hgt ~ age) %>%
        coef())
```

map() ON LARGE LISTS

```r
# how many samples?
length(samples)
```

```
[1] 1000
```

```r
# what is the first sample?
samples[[1]] %>% slice_head()
```

```
        age   hgt  wgt   bmi hc  gen  phb tv  reg
4501 12.342 158.1 54.9 21.96 NA <NA> <NA> NA west
```

```r
# what is the first sample's linear model?
samples_lm[[1]]
```

```
(Intercept)         age 
  70.975166    6.573117 
```

```r
# what are the first three samples' linear model?
samples_lm[1:3]
```

```
[[1]]
(Intercept)         age 
  70.975166    6.573117 

[[2]]
(Intercept)         age 
  69.902756    6.687102 

[[3]]
(Intercept)         age 
  70.595459    6.583677 
```

Reduce()

With `reduce()`, you can combine the results obtained with `map()`.

```r
reduce(samples_lm, `+`) # sum of lm coefficients
```

```
(Intercept)         age 
  70681.662    6594.446 
```

```r
reduce(samples_lm, `+`) / 1000 # average lm coefficients
```

```
(Intercept)         age 
  70.681662    6.594446 
```

STRUCTURING THE OUTPUT OF `map()`

`map()` returns a list, which can be structured into a data frame using `map_df()`. This is useful when you want to convert the results of `map()` into a tidy format.

```r
# with map_df() instead of map() to return a data frame
samples %>%
  map_df(~.x %$%
           lm(hgt ~ age) %>%
           coef())
```

```
# A tibble: 1,000 × 2
   `(Intercept)`   age
           <dbl> <dbl>
 1          71.0  6.57
 2          69.9  6.69
 3          70.6  6.58
 4          70.8  6.56
 5          70.2  6.56
 6          70.2  6.67
 7          70.8  6.67
 8          72.2  6.48
 9          69.7  6.71
10          70.9  6.57
# ℹ 990 more rows
```

STRUCTURING THE OUTPUT OF `map()`

```r
# with map_df() if an object already exists as a list
samples_lm %>%
  map_df(~tibble(intercept = .x[1], slope = .x[2]))
```

```
# A tibble: 1,000 × 2
   intercept slope
       <dbl> <dbl>
 1      71.0  6.57
 2      69.9  6.69
 3      70.6  6.58
 4      70.8  6.56
 5      70.2  6.56
 6      70.2  6.67
 7      70.8  6.67
 8      72.2  6.48
 9      69.7  6.71
10      70.9  6.57
# ℹ 990 more rows
```

FUTURES

- The `future` package enables **asynchronous** and **parallel** processing in R.
- It allows R to perform tasks **in the background**, freeing up your current R session.
- Ideal for:
  - Speeding up long-running computations
  - Running tasks concurrently

WHY USE `future`?

- Normally, R runs code **line-by-line** (sequentially).
- `future` lets you run tasks **in parallel**, improving efficiency.
- Example use cases:
  - Simulations
  - Data processing across multiple cores
  - Web scraping multiple pages

future_map()

The `future_map()` function is part of the `furrr` package, which integrates the `future` package with the `purrr` package's mapping functions.

It allows you to apply a function to each element of a list or vector in parallel, making it easier to handle large datasets or computationally intensive tasks.

```
# Set up parallel processing
plan(multisession) # Use multiple cores for parallel processing
samples %>%
  future_map_dfr(~.x %$%
           lm(hgt ~ age) %>%
           coef())
```

```
# A tibble: 1,000 × 2
   `(Intercept)`   age
           <dbl> <dbl>
 1          71.0  6.57
 2          69.9  6.69
 3          70.6  6.58
 4          70.8  6.56
 5          70.2  6.56
 6          70.2  6.67
 7          70.8  6.67
 8          72.2  6.48
 9          69.7  6.71
10          70.9  6.57
# ℹ 990 more rows
```

```
plan(sequential) # Stop parallel processing and reset to sequential processing
```

future_map()

The `future_map()` function is part of the `furrr` package, which integrates the `future` package with the `purrr` package's mapping functions.

It allows you to apply a function to each element of a list or vector in parallel, making it easier to handle large datasets or computationally intensive tasks.

```r
# Set up parallel processing
plan(multisession) # Use multiple cores for parallel processing
samples %>%
  future_map_dfc(~.x %$%
           lm(hgt ~ age) %>%
           coef())
```

```
# A tibble: 2 × 1,000
   ...1  ...2  ...3  ...4  ...5  ...6  ...7  ...8  ...9 ...10 ...11 ...12 ...13
  <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
1 71.0  69.9  70.6  70.8  70.2  70.2  70.8  72.2  69.7  70.9  70.3  70.5  69.5 
2  6.57  6.69  6.58  6.56  6.56  6.67  6.67  6.48  6.71  6.57  6.61  6.55  6.68
# ℹ 987 more variables: ...14 <dbl>, ...15 <dbl>, ...16 <dbl>, ...17 <dbl>,
#   ...18 <dbl>, ...19 <dbl>, ...20 <dbl>, ...21 <dbl>, ...22 <dbl>,
#   ...23 <dbl>, ...24 <dbl>, ...25 <dbl>, ...26 <dbl>, ...27 <dbl>,
#   ...28 <dbl>, ...29 <dbl>, ...30 <dbl>, ...31 <dbl>, ...32 <dbl>,
#   ...33 <dbl>, ...34 <dbl>, ...35 <dbl>, ...36 <dbl>, ...37 <dbl>,
#   ...38 <dbl>, ...39 <dbl>, ...40 <dbl>, ...41 <dbl>, ...42 <dbl>,
#   ...43 <dbl>, ...44 <dbl>, ...45 <dbl>, ...46 <dbl>, ...47 <dbl>, …
```

```r
plan(sequential) # Stop parallel processing and reset to sequential processing
```

BINAIRY OPERATORS LIKE `%in%` AND `%>%`

HOW BINAIRY OPERATORS WORK

Binary operators are functions that take two arguments and return a single value. In R, you can create your own binary operators using the `"%operator%"` syntax.

```r
`%my_operator%` <- function(x, y) {
  # perform some operation on x and y
  result <- x + y  # example operation: addition
  return(result)
}

4 %my_operator% 5
```

```
[1] 9
```

```r
1:4 %my_operator% 5:8
```

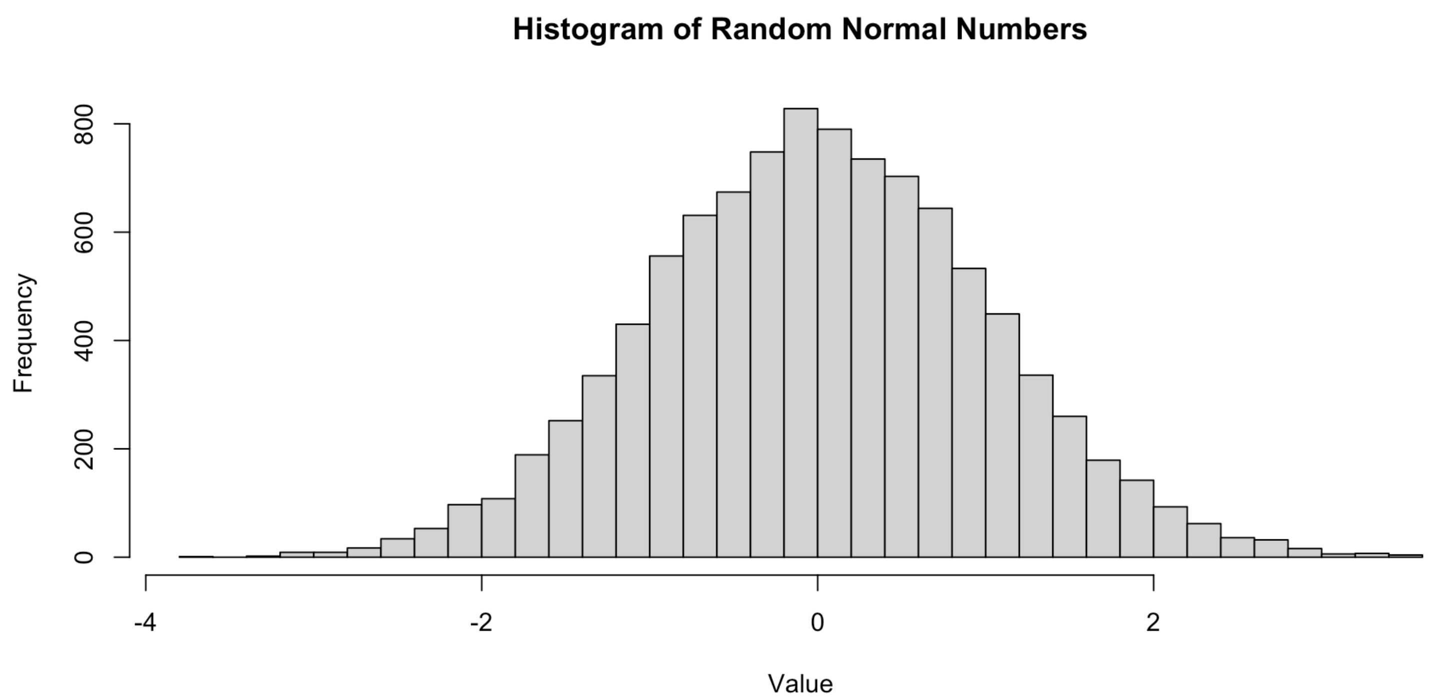
```
[1]  6  8 10 12
```

Binary operators allow you to write the function in a more natural way, similar to mathematical notation. You can use them for various operations, such as addition, subtraction, multiplication, or any custom operation you define.

DRAWING FROM DISTRIBUTIONS

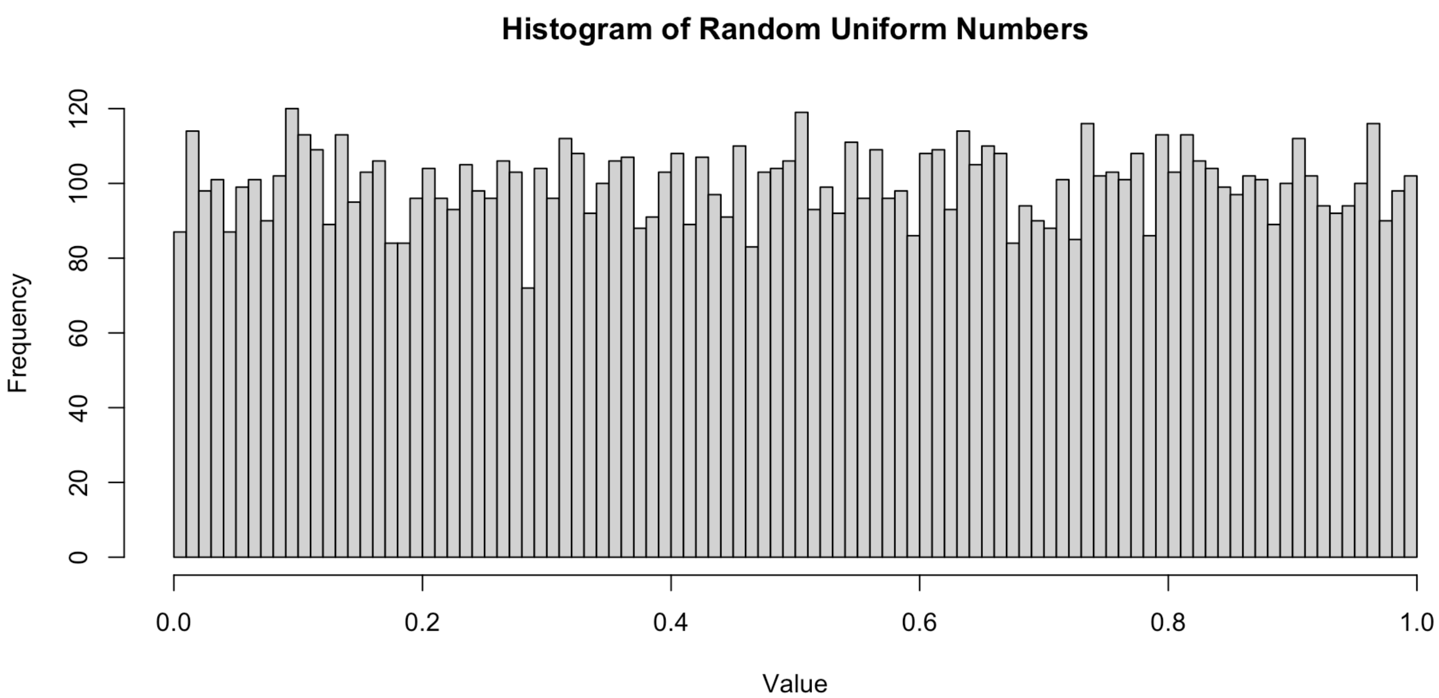
NORMAL DISTRIBUTION

```r
# Draw 10000 random numbers from a normal distribution
normals <- rnorm(10000, mean = 0, sd = 1)
# Plot the histogram of the random numbers
hist(normals,
     breaks = 30,
     main = "Histogram of Random Normal Numbers",
     xlab = "Value",
     ylab = "Frequency")
```



UNIFORM DISTRIBUTION

```r
# Draw 10000 random numbers from a uniform distribution
uniforms <- runif(10000, min = 0, max = 1)
# Plot the histogram of the random numbers
hist(uniforms,
     breaks = 80,
     main = "Histogram of Random Uniform Numbers",
     xlab = "Value",
     ylab = "Frequency")
```



RANDOM NUMBER GENERATORS

HOW PRNGS WORK

Pseudo Random Number Generators (PRNGs) are algorithms that generate sequences of numbers that approximate the properties of random numbers. They are called "pseudo" because they are deterministic and can be reproduced if the initial state (seed) is known.

```r
# fix the seed
set.seed(123)
# draw 10 random integers between 1 and 100 without replacement
sample(1:100, size = 10, replace = FALSE)
```

```
[1] 31 79 51 14 67 42 50 43 97 25
```

```r
# fix the seed again
set.seed(123)
# draw 10 random integers between 1 and 100 without replacement
sample(1:100, size = 10, replace = FALSE)
```

```
[1] 31 79 51 14 67 42 50 43 97 25
```

```r
# draw the same 10 random numbers in sets of 5
set.seed(123)
sample(1:100, size = 5, replace = FALSE)
```

```
[1] 31 79 51 14 67
```

```r
sample(1:100, size = 5, replace = FALSE)
```

```
[1] 42 50 43 14 25
```

BEWARE: once you fix the random seed, everything that uses random numbers will become seed-dependent. Your findings can be accidental. Always replicate with another seed!

HOW PRNGS WORK

Pseudo Random Number Generators (PRNGs) are algorithms that generate sequences of numbers that approximate the properties of random numbers. They are called “pseudo” because they are deterministic and can be reproduced if the initial state (seed) is known.

```
# draw 10 numbers
set.seed(123)
rnorm(10)
```

```
 [1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774  1.71506499
 [7]  0.46091621 -1.26506123 -0.68685285 -0.44566197
```

```
# draw 10 numbers in sets of 5
set.seed(123)
rnorm(5)
```

```
[1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774
```

```
rnorm(5)
```

```
[1]  1.7150650  0.4609162 -1.2650612 -0.6868529 -0.4456620
```

```
# draw 15 numbers in two sets, where the first set is 5 numbers
set.seed(123)
rnorm(5)
```

```
[1] -0.56047565 -0.23017749  1.55870831  0.07050839  0.12928774
```

```
rnorm(10)
```

```
 [1]  1.7150650  0.4609162 -1.2650612 -0.6868529 -0.4456620  1.2240818
 [7]  0.3598138  0.4007715  0.1106827 -0.5558411
```

REPLICATION VS REPRODUCTION

Reproduction is the process of running the same analysis with the same data and code to see if the results can be exactly replicated.

```r
# reproduction
set.seed(123)
rnorm(10) %>% mean()
```

```
[1] 0.07462564
```

```r
set.seed(123)
rnorm(10) %>% mean()
```

```
[1] 0.07462564
```

Replication is the process of running the same analysis on a different dataset or in a different context to see if the results are consistent.

```r
# replication WITH reproduction
set.seed(123)
rnorm(10) %>% mean()
```

```
[1] 0.07462564
```

```r
set.seed(124)
rnorm(10) %>% mean()
```

```
[1] 0.2147669
```

PRACTICAL