# *6e.* Illustration - Johnny, the YouTuber

Martin Alfaro
PhD in Economics

## INTRODUCTION

Through a practical example, this section will demonstrate the convenience of the following features:

1. Boolean indexing for working with subsets of the data
2. organizing code around functions
3. pipes to enhance code readability
4. use of views to modify subsets of the data

## DESCRIBING THE SCENARIO

We'll explore the stats of Johnny's YouTube channel during a month. He has a median of 50,000 visits per video, with a few viral videos exceeding 100,000 visits. The information at our disposal is:

- `nr_videos`: 30 (one per day).
- `visits`: viewers per video (in thousands).
- `payrates`: Dollars paid per video for 1,000 visits. They range from $2 to $6. The fluctuation is consistent with YouTube's payment model, which depends on a video's feature (e.g., content, duration, retention).

The scenario is modeled by some mock data. The details of how data are generated are unimportant, but were added below for the sake of completeness. Ultimately, what matters is that the mock data creates the variables `visits` and `payrates`.

```
using StatsBase, Distributions
using Random; Random.seed!(1234)

function audience(nr_videos; median_target)
    shape = log(4,5)
    scale = median_target / 2^(1/shape)

    visits = rand(Pareto(shape,scale),  nr_videos)

    return visits
end

nr_videos = 30

visits   = audience(nr_videos, median_target = 50)      # in thousands of visits
payrates = rand(2:6, nr_videos)                         # per thousands of visits
```

```
julia> visits # in thousands
30-element Vector{Float64}:
 38.8086
 70.8113
  ⋮
 72.3673
 30.2565
julia> payrates # per thousand visits
30-element Vector{Int64}:
 5
 3
 ⋮
 2
 4
```

The variables `visits` and `payrates` enable us to calculate the total payment per video.

```
earnings = visits .* payrates
```

```
julia> earnings
30-element Vector{Float64}:
 194.043
 212.434
    ⋮
 144.735
 121.026
```

## SOME GENERAL INFORMATION

We begin by examining the per-view payments made by YouTube. We first show that Johnny's payments range from \$2 to \$6. Moreover, using the `countmaps` function from the `StatsBase` package, we conclude that Johnny has eight videos reaching the maximum payment of \$6.

```
range_payrates  = unique(payrates) |> sort
```

```
julia> range_payrates
5-element Vector{Int64}:
 2
 3
 4
 5
 6
```

```
using StatsBase
occurrences_payrates = countmap(payrates) |> sort
```

```
julia> occurrences_payrates
OrderedDict{Int64, Int64} with 5 entries:
  2 => 5
  3 => 6
  4 => 8
  5 => 5
  6 => 6
```

We can also provide some insights into Johnny's most profitable videos. By applying the `sort` function, we can isolate his top 3 highest-earning videos. Moreover, we can apply the `sortperm` function to identify their indices, allowing us to extract the payment per view and total visits associated with each video.

```
top_earnings    = sort(earnings, rev=true)[1:3]
```

```
julia> top_earnings
3-element Vector{Float64}:
 7757.81
  693.813
  672.802
```

```
indices         = sortperm(earnings, rev=true)[1:3]

sorted_payrates = payrates[indices]
```

```
julia> sorted_payrates
3-element Vector{Int64}:
 6
 6
 6
```

```
indices         = sortperm(earnings, rev=true)[1:3]

sorted_visits   = visits[indices]
```

```
julia> sorted_visits
3-element Vector{Float64}:
 1292.97
  115.636
  112.134
```

## BOOLEAN VARIABLES

In the following, we demonstrate how to use Boolean indexing to extract and characterize subsets of data. Our focus will be on characterizing Johnny's viral videos, defined as those that have surpassed a threshold of 100k visits. In particular, we'll determine the number of visits and revenue generated by them.

To identify the viral videos, we'll create a `Bool` vector, where `true` identifies a viral video. This vector can then be employed as a logical index, allowing us to selectively extract data points from other variables. In the example below, we apply it to compute the total visits and earnings attributable to the viral videos.

```
# characterization of viral videos
viral_threshold = 100
is_viral        = (visits .≥ viral_threshold)

# stats
viral_nrvideos  = sum(is_viral)
viral_visits    = sum(visits[is_viral])
viral_revenue   = sum(earnings[is_viral])
```

```
julia> viral_nrvideos
4
julia> viral_visits
1625.05
julia> viral_revenue
9750.3
```

Boolean indexing also enables subsetting data satisfying multiple conditions. For instance, we can apply this technique to calculate the proportion of viral videos for which YouTube paid more than $3 per thousand visits.

```
# characterization
viral_threshold    = 100
payrates_above_avg = 3

is_viral           = (visits .≥ viral_threshold)
is_viral_lucrative = (visits .≥ viral_threshold) .&& (payrates .> payrates_above_avg)

# stat
proportion_viral_lucrative = sum(is_viral_lucrative) / sum(is_viral) * 100
```

```
julia> proportion_viral_lucrative
100.0
```

**Rounding Outputs**

You can express results with rounded numbers via the function `round`. By default, this returns the nearest integer expressed as a `Float64` number.

The function also offers additional specifications. For instance, the number of decimal places in the approximation can be controlled via the `digits` keyword argument. Furthermore, it's possible to represent the number as an integer using either `Int` or `Int64` as an argument. [1]

```
rounded_proportion = round(proportion_viral_lucrative)
```

```
julia> rounded_proportion
100.0
```

```
rounded_proportion = round(proportion_viral_lucrative, digits=1)
```

```
julia> rounded_proportion
100.0
```

```
rounded_proportion = round(Int64, proportion_viral_lucrative)
```

```
julia> rounded_proportion
100
```

## FUNCTIONS TO REPRESENT TASKS

The approach employed so far allows for a quick exploration of Johnny's viral videos. However, it lacks the structure needed for a systematic analysis across different subsets of the data. To address this limitation, we can capture the characterization of videos through a function.

Recall that a well-designed function should embody a single clearly defined task. In our case, the goal is to subset data and extract key statistics, including the number of videos, visits, and revenue generated. Furthermore, the function should remain independent of any specific application, so it can be reused to analyze different groups of videos without rewriting code each time.

The function below implements this task taking three arguments: the raw data (`visits` and `payrates`) and a condition that defines the subset of data (`condition`). By keeping the condition generic, the function is flexible enough to target any subset of videos. The example also showcases the convenience of pipes to compute intermediate temporary steps.

```
#
function stats_subset(visits, payrates, condition)
    nrvideos = sum(condition)
    audience = sum(visits[condition])

    earnings = visits .* payrates
    revenue  = sum(earnings[condition])

    return (; nrvideos, audience, revenue)
end
```

```
using Pipe
function stats_subset(visits, payrates, condition)
    nrvideos = sum(condition)
    audience = sum(visits[condition])


    revenue  = @pipe (visits .* payrates) |> x -> sum(x[condition])

    return (; nrvideos, audience, revenue)
end
```

```
using Pipe
function stats_subset(visits, payrates, condition)
    nrvideos = sum(condition)
    audience = sum(visits[condition])


    revenue  = @pipe (visits .* payrates) |> sum(_[condition])

    return (; nrvideos, audience, revenue)
end
```

Below, we demonstrate the reusability of the function by characterizing various subsets of data.

```
viral_threshold  = 100
is_viral         = (visits .≥ viral_threshold)
viral            = stats_subset(visits, payrates, is_viral)
```

```
julia> viral
(nrvideos = 4, audience = 1625.05, revenue = 9750.3)
```

```
viral_threshold  = 100
is_notviral      = .!(is_viral)      # '!' is negating a boolean value and we broadcast it
notviral         = stats_subset(visits, payrates, is_notviral)
```

```
julia> notviral
(nrvideos = 26, audience = 1497.02, revenue = 5687.67)
```

```
days_to_consider = (1, 10, 25)      # subset of days to be characterized
is_day           = in.(eachindex(visits), Ref(days_to_consider))
specific_days    = stats_subset(visits, payrates, is_day)
```

```
julia> specific_days
(nrvideos = 3, audience = 182.939, revenue = 1030.33)
```

## VARIABLE MUTATION

Suppose Johnny is exploring ways to increase viewership through targeted advertising. His projections suggest that ads will boost viewership per video by 20%. However, due to budget constraints, Johnny must choose between promoting either his non-viral or viral ones. To make an informed decision, Johnny decides to leverage the data at his disposal to crunch some rough estimates. In particular, he'll base his decision on the earnings he would've earned if he had run targeted ads.

The first step in this process involves creating a modified copy of `visits`. This should now reflect the anticipated increase in viewership from running ads on the targeted videos (either viral or non-viral). With this updated audience data, Johnny can then apply the previously defined `stats_subset` function to estimate the potential earnings. By comparing the estimations for each group of targeted video, Johnny can determine which strategy offers the higher return on investment.

```
# 'temp' modifies 'new_visits'
new_visits      = copy(visits)
temp            = @view new_visits[new_visits .≥ viral_threshold]
temp           .= 1.2 .* temp

allvideos       = trues(length(new_visits))
targetViral     = stats_subset(new_visits, payrates, allvideos)
```

```
julia> targetViral
(nrvideos = 30, audience = 3447.08, revenue = 17388.0)
```

```
# 'temp' modifies 'new_visits'
new_visits      = copy(visits)
temp            = @view new_visits[new_visits .< viral_threshold]
temp           .= 1.2 .* temp

allvideos       = trues(length(new_visits))
targetNonViral  = stats_subset(new_visits, payrates, allvideos)
```

```
julia> targetNonViral
(nrvideos = 30, audience = 3421.47, revenue = 16575.5)
```

Given the results in each tab, promoting viral videos appears to be the more profitable option.

**Be Careful with Misusing 'view'**

Updating `temp` requires an in-place operation to mutate the parent object. In our case, this was achieved via the broadcasted operator `.=`. Below, we present some implementations that fail to produce the intended result.

```
new_visits = copy(visits)


temp  = @view new_visits[new_visits .≥ viral_threshold]
temp .= temp .* 1.2
```

```
new_visits = visits      # it creates an alias, it's a view of
the original object!!!

# 'temp' modifies 'visits' -> you lose the original info
temp  = @view new_visits[new_visits .≥ viral_threshold]
temp .= temp .* 1.2
```

```
new_visits = copy(visits)

# wrong -> not using `temp .= temp .* 1.2`
temp  = @view new_visits[new_visits .≥ viral_threshold]
temp  = temp .* 1.2      # it creates a new variable 'temp', it
does not modify 'new_visits'
```

**Use of "Let Blocks" To Avoid Bugs**

In the code above, "Target Viral" and "Target Non-Viral" reference variables with identical names. This creates the risk of accidentally referring to a variable from the wrong scenario.

A practical way to mitigate this risk is by employing "let blocks". Since each let block introduces its own scope, this helps maintain a clean namespace and prevents variable collisions.

```
targetViral      = let visits = visits, payrates = payrates,
threshold = viral_threshold
    new_visits = copy(visits)
    temp       = @view new_visits[new_visits .≥ threshold]
    temp      .= 1.2 .* temp

    allvideos  = trues(length(new_visits))
    stats_subset(new_visits, payrates, allvideos)
end
```

```
julia> targetViral
(nrvideos = 30, audience = 3447.08, revenue = 17388.0)
```

```
targetNonViral  = let visits = visits, payrates = payrates,
threshold = viral_threshold
    new_visits = copy(visits)
    temp       = @view new_visits[new_visits .< threshold]
    temp      .= 1.2 .* temp

    allvideos  = trues(length(new_visits))
    stats_subset(new_visits, payrates, allvideos)
end
```

```
julia> targetNonViral
(nrvideos = 30, audience = 3421.47, revenue = 16575.5)
```

## BROADCASTING OVER A LIST OF FUNCTIONS

A function like `stats_subset` is useful for computing a fixed set of summary statistics. However, since the choice of statistics is hard-coded into the function's definition, the output can't be changed without rewriting the code. This rigidity makes the function less reusable across different analytical contexts.

A more flexible approach consists of specifying which statistics to compute at the time of use. Julia makes this possible because functions are *first-class objects*, entailing that functions behave just like any other variable. This feature lets us define a list of statistical functions, eventually applying them element-wise to the variables we want to characterize.

Below, we apply this methodology to characterize the variable `visits`.

```
list_functions = [sum, median, mean, maximum, minimum]

stats_visits   = [fun(visits) for fun in list_functions]
```

```
julia> stats_visits
5-element Vector{Float64}:
 3447.08
   64.8765
  114.903
 1551.56
   28.2954
```

The same methodology can also be employed for characterizing multiple variables at once. In fact, broadcasting makes this straightforward to implement. For instance, below we simultaneously characterize `visits` and `earnings`.

```
list_functions = [sum, median, mean, maximum, minimum]

stats_various  = [fun.([visits, payrates]) for fun in list_functions]
```

```
julia> stats_various
5-element Vector{Vector{Float64}}:
 [3447.08, 121.0]
 [64.8765, 4.0]
 [114.903, 4.03333]
 [1551.56, 6.0]
 [28.2954, 2.0]
```

One major limitation of the current method is its inability to reflect each statistic's name. To address this, we can collect all statistics in a named tuple, enabling the access of each through its name. For instance, given a named tuple `stats_visits`, it'll become possible to retrieve the average value of `visits` by `stats_visits.mean` or `stats_visits[:mean]`.

To assign names to the statistics within the named tuple, we'll use the `Symbol` type. This translates strings into identifiers that can act as keys of a named tuple, enabling programmatic access to each statistic.

```
vector_of_tuples = [(Symbol(fun), fun(visits)) for fun in list_functions]
stats_visits     = NamedTuple(vector_of_tuples)
```

```
julia> stats_visits
(sum = 3447.08, median = 64.8765, mean = 114.903, maximum = 1551.56, minimum = 28.2954)
julia> stats_visits.mean
114.903
julia> stats_visits[:median]
64.8765
```

**FOOTNOTES**

[1.] Recall that the type `Int` defaults to `Int64` on 64-bit systems and to `Int32` for 32-bit systems. Most modern computers fall into the former category, explaining why we usually employ `Int64`.