6e. Illustration - Johnny, the YouTuber

Martin Alfaro
PhD in Economics

INTRODUCTION

Through this section's illustration, we'll show 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, ranging 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 they were added below for the sake of completeness. What matters is that the mock data creates the following two variables.

```
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}:
 44.4608
 57.2323
  ⋮
 86.4182
 36.5051
julia> payrates # per thousand visits
30-element Vector{Int64}:
 2
 6
 ⋮
 3
 4
```

These two variables enable us to calculate the total payment per video.

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

```
julia> earnings
30-element Vector{Float64}:
  88.9215
 343.394
   ⋮
 259.254
 146.02
```

SOME GENERAL INFORMATION

We begin by presenting some information about the per-view payments made by YouTube. We first confirm that Johnny's payments ranged 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 => 4
  3 => 5
  4 => 8
  5 => 5
  6 => 8
```

We can also provide some insights into Johnny's most profitable videos. By applying the `sort` function and isolating the top 3 videos, we can obtain information on the highest earnings videos. Moreover, we can apply the `sortperm` function to identify the indices of these videos, allowing us to extract the payment per view and total visits associated with each one.

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

```
julia> top_earnings
3-element Vector{Float64}:
 2708.57
 1083.07
  723.493
```

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

sorted_payrates = payrates[indices]
```

```
julia> sorted_payrates
3-element Vector{Int64}:
 6
 5
 5
```

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

sorted_visits   = visits[indices]
```

```
julia> sorted_visits
3-element Vector{Float64}:
 451.428
 216.615
 144.699
```

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 these videos.

To identify the viral videos, we'll create a `Bool` vector, where `true` identifies a viral video. This vector allows us to selectively extract data points from other variables by using them as indices. For instance, we use it below to compute the total visits and earnings derived from these 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
6
julia> viral_visits
1243.63
julia> viral_revenue
6133.19
```

Boolean indexing also makes it possible to subset data satisfying multiple conditions. For instance, we can use 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
83.3333
```

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, you can control the number of decimal places in the approximation using the `digits` keyword argument. Furthermore, it's possible to represent the number as an `Int64` using the argument `Int`.

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

```
julia> rounded_proportion
83.0
```

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

```
julia> rounded_proportion
83.3
```

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

```
julia> rounded_proportion
83
```

FUNCTIONS TO REPRESENT TASKS

The approach employed so far allows for a rapid exploration of Johnny's viral videos. However, it falls short in providing a systematic analysis that could be extended to other subsets of data. To overcome this limitation, we can automate the process by defining a function.

Recall that a well-designed function should embody a specific task, implying that it must be independent of its specific application. In our case, the goal is to subset data and extract specific statistics, including the number of videos, visits, and revenue generated.

The function below implements this task taking three arguments: the raw data (`visits` and `payrates`) and a condition defining the subset (`condition`). By keeping the condition generic, we can seamlessly apply the same analysis to various subsets of data. The example also showcases the convenience of pipes to compute intermediate temporary steps, using it to retrieve the income earned from a subset of videos.

```
#
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 illustrate how the function enables effortlessly characterizing various subsets of data.

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

```
julia> viral
(nrvideos = 6, audience = 1243.63, revenue = 6133.19)
```

```
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 = 24, audience = 1169.13, revenue = 4971.02)
```

```
days_to_consider = (1, 10, 25)      # days when the videos were posted
is_day           = in.(eachindex(visits), Ref(days_to_consider))
specific_days    = stats_subset(visits, payrates, is_day)
```

```
julia> specific_days
(nrvideos = 3, audience = 118.547, revenue = 414.113)
```

MUTATING VARIABLES

Suppose that, seeking to enhance audience engagement, Johnny has decided to promote his videos through 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 videos or his 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 during last month if he had run targeted ads.

The computations require creating a modified copy of `visits`, adjusting the audience data after running the ads for the targeted videos. With this updated audience data, we can then apply the previously defined `stats_subset` function to estimate the potential earnings. Comparing the results in each tab, Johnny would conclude that promoting viral videos seems to be a more profitable strategy.

```
# '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 = 2661.48, revenue = 12330.8)
```

```
# '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 = 2646.58, revenue = 12098.4)
```

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 state some implementations that fail to produce the desired 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

The code above for "Target Viral" and "Target Non-Viral" refers to each variable by an identical name. This increases the risk of accidentally referring to a variable from a different scenario.

The likelihood of incurring this issue can be alleviated by employing "let blocks". By defining their own scope, they help maintain a clean namespace.

```
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 = 2661.48, revenue = 12330.8)
```

```
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 = 2646.58, revenue = 12098.4)
```

BROADCASTING OVER A LIST OF FUNCTIONS (*OPTIONAL*)

At the beginning of the analysis, we could've derived descriptive statistics to gain insights about Johnny's videos. This can be accomplished by using the `describe` function from the `StatsBase` package. Despite this, the presentation aims to highlight that functions are first-class objects in Julia. This property entails that a function behaves just like any other variable, allowing the user to define a list of functions and then apply them element-wise to a variable.

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

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

```
julia> stats_visits
5-element Vector{Float64}:
 2661.48
   52.7884
   88.716
  541.714
   27.7205
```

By broadcasting the operation, we can also compute stats for multiple variables concurrently. For instance, below we characterize `visits` and `earnings` simultaneously.

```
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}}:
 [2661.48, 128.0]
 [52.7884, 4.0]
 [88.716, 4.26667]
 [541.714, 6.0]
 [27.7205, 2.0]
```

One major limitation of the current method is its inability to capture each statistic's name. To overcome this, we can employ a named tuple, which we'll call `stats_visits`. The approach enables us to access stats through their respective names, such as `stats_visits.mean` or `stats_visits[:mean]` for the average value.

The implementation is based on the type `Symbol`. This converts strings into identifiers, which are necessary to programmatically access a named tuple's keys.

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

```
julia> stats_visits
(sum = 2661.48, median = 52.7884, mean = 88.716, maximum = 541.714, minimum = 27.7205)
julia> stats_visits.mean
88.716
julia> stats_visits[:median]
52.7884
```