FAF.PTR16.1 -- Project 0

> **Performed by:** Munteanu Dumitru, group FAF-202 **Verified by:** asist. univ. Alexandru Osadcenco

P0W1

Task 1 -- Install elixir **Task 2** -- Write a script that would print the message “Hello PTR” on the screen

```
defmodule  HelloWorld  do
    def  hello() do
        "Hello PTR"
    end
end
```

This code defines a module that has a function which returns the string "Hello PTR".

Task 3 -- Initialize a repository https://github.com/dumitrumunteanu/ptr

Task 4 -- Write a readme for the repository This can be seen inside the repo.

Task 5 -- Write a unit test

```
defmodule  HelloWorldTest  do
    use  ExUnit.Case
    doctest HelloWorld

    test "greets the world"  do
        assert HelloWorld.hello() == "Hello PTR"
    end
end
```

P0W2

Task 1 -- Write a function that determines whether an input integer is prime

```
def isPrime(x) do
    cond do
      x < 2 -> false
      x >= 2 ->
        from = 2
```

```
        to = trunc(:math.sqrt(x))
        n_total = to - from + 1

        n_tried =
          Enum.take_while(from..to, fn i -> rem(x, i) != 0 end)
          |> Enum.count()

        n_total == n_tried
    end
  end
```

This function checks if a given integer is a prime number by iterating over a range of integers from 2 to the square root of the input, and checking if any of those integers are factors of the input. If all integers in the range are tried and none of them are factors of the input, the function returns `true`, indicating that the input is a prime number. Otherwise, the function returns `false`.

Task 2 -- Write a function to calculate the area of a cylinder, given it's height and radius.

```
  def cylinderArea(height, radius) do
    baseArea = :math.pi() * radius * radius
    sideArea = 2 * :math.pi() * radius * height

    sideArea + 2 * baseArea
  end
```

This function translates the mathematical formula for the cylinder area into elixir.

Task 3 -- Write a function to reverse a list.

```
def reverse(list) do
    Enum.reverse(list)
  end
```

This function uses Enum.reverse to reverse a list.

Task 4 -- Write a function to calculate the sum of unique elements in a list.

```
def uniqueSum(list) do
    Enum.sum(Enum.uniq(list))
  end
```

Pretty self-explanatory

Task 5 -- Write a function that extracts a given number of randomly selected elements from a list.

```
def extractRandomN(list, n) do
    list |> Enum.shuffle |> Enum.take(n)
  end
```

this function takes a list and a number `n` as inputs, shuffles the elements in the list randomly, and returns the first `n` elements of the shuffled list as a new list. This function can be useful in situations where a random sample of elements from a larger list is needed, such as in data analysis or machine learning applications.

Task 6 -- Write a function that returns the first n elements of the Fibonacci sequence.

```
def fib(n) do
    cond do
      n <= 2 -> 1
      n > 2 -> fib(n - 1) + fib(n - 2)
    end
  end

def firstFibonacciElements(n) do
    Enum.map(1..n, fn i -> Tasks.fib(i) end)
  end
```

This function is implemented on the definition of the fibonacci sequence

Taks 8 -- Write a function that, given a dictionary, would translate a sentence. Words not found in the dictionary need not be translated.

```
def translator(dictionary, originalString) do
    originalString
      |> String.split(" ")
      |> Enum.map(fn word ->
        if Map.has_key?(dictionary, word) do
          dictionary[word]
        else
          word
        end
      end)
      |> Enum.join(" ")
  end
```

this function takes a dictionary map and a string as inputs, and translates words in the string using the dictionary map. The function splits the input string into a list of words, maps over the list to translate each word using the dictionary, and then joins the resulting list of translated or original words back into a string. This function can be useful in situations where automatic translation of text is needed, such as in language learning or communication tools.

Task 9 -- Write a function that receives as input three digits and arranges them in an order that would create the smallest possible number. Numbers cannot start with a 0.

```
def smallestNumber(a, b, c) do
    digitList = [a, b, c]
    orderedDigits = digitList |> Enum.sort()
    firstNonZeroDigitIdx = orderedDigits |> Enum.find_index(fn digit -> digit != 0 end)

    Enum.join(
      [Enum.at(orderedDigits, firstNonZeroDigitIdx)] ++
      List.delete_at(orderedDigits, firstNonZeroDigitIdx)
    )
  end
```

this function takes three digits as inputs, sorts them in ascending order, extracts the smallest non-zero digit, and concatenates it with the remaining digits to form the smallest possible number. This function can be useful in situations where it is necessary to determine the minimum value that can be formed using a set of digits, such as in numerical algorithms or coding challenges.

Task 10 -- Write a function that, given an array of strings, will return the words that can be typed using only one row of the letters on an English keyboard layout.

```
  def lineWords(wordList) do
    wordList
      |> Enum.filter(fn word ->
        row1 = "qwertyuiop"
        row2 = "asdfghjkl"
        row3 = "zxcvbnm"

        word
          |> String.downcase()
          |> String.split("")
          |> Enum.all?(fn letter -> String.contains?(row1, letter) end)
        or
        word
          |> String.downcase()
          |> String.split("")
          |> Enum.all?(fn letter -> String.contains?(row2, letter) end)
```

```
          or
          word
            |> String.downcase()
            |> String.split("")
            |> Enum.all?(fn letter -> String.contains?(row3, letter) end)
        end)
    end
```

this function takes a list of words as input, and filters the list to include only those words that can be typed using a single row of a standard keyboard. This function can be useful in situations where it is necessary to filter a large list of words based on a specific condition, such as in natural language processing or data cleaning tasks.

Task 11 -- White a function that, given an array of strings, would group the anagrams together.

```
  def groupAnagrams(strings) do
      Enum.group_by(strings, fn string ->
        string
        |> String.graphemes()
        |> Enum.sort()
      end)
      |> Enum.map(fn {key, value} ->
        { key |> Enum.join(""), value |> Enum.sort() }
        # value |> Enum.sort()
      end)
    end
```

this function takes a list of strings as input, and groups the strings into anagrams based on the sorted list of graphemes in each string.

P0W3

I do not get it why we should waste time on writing reports if we present the labs.

P0W4

Task -- Create a supervised processing line to clean messy strings. The first worker in the line would split the string by any white spaces (similar to Python's str.split method). The second actor will lowercase all words and swap all m's and n's (you nomster!). The third actor will join back the sentence with one space between words (similar to Python's str.join method). Each worker will receive as input the previous actor's output, the last actor printing 5 the result on screen. If any of the workers die because it encounters an error, the whole processing line needs to be restarted. Logging is welcome

```
defmodule Splitter do
  def start_link() do
    pid = spawn_link(__MODULE__, :loop, [])
    IO.puts("A splitter process has started at #{inspect(pid)}.")
    {:ok, pid}
  end

  def loop() do
    receive do
      {string} ->
        strings = string |> String.split()

        IO.puts("\"#{string}\" was split into #{inspect(strings)}")

        {_, swapper, _, _} = FormatterSupervisor |> Supervisor.which_children |> Enum.at(
        send(swapper, {strings})

        loop()
    end
  end
end

defmodule Swapper do
  def start_link() do
    pid = spawn_link(__MODULE__, :loop, [])
    IO.puts("A swapper process has started at #{inspect(pid)}.")
    {:ok, pid}
  end

  def loop() do
    receive do
      {strings} ->
        transformed_strs = strings |> Enum.map(fn string ->
          string
          |> String.downcase
          |> String.replace("m", "__temp__")
          |> String.replace("n", "m")
          |> String.replace("__temp__", "n")
        end)

        IO.puts("\"#{inspect(strings)}\" was transformed into #{inspect(transformed_strs)

        {_, joiner, _, _} = FormatterSupervisor |> Supervisor.which_children |> Enum.at(0
        send(joiner, {transformed_strs})

        loop()
    end
  end
```

```
  end

  defmodule Joiner do
    def start_link() do
      pid = spawn_link(__MODULE__, :loop, [])
      IO.puts("A joiner process has started at #{inspect(pid)}.")
      {:ok, pid}
    end

    def loop() do
      receive do
        {strings} ->
          joined_list = strings |> Enum.join(" ")

          IO.puts("#{inspect(strings)} joined into \"#{joined_list}\"")

          loop()
      end
    end
  end

  defmodule FormatterSupervisor do
    use Supervisor

    def start_link() do
      Supervisor.start_link(__MODULE__, :ok, name: __MODULE__)
    end

    @impl true
    def init(:ok) do
      children = [
        %{
          id: :splitter,
          start: {Splitter, :start_link, []}
        },
        %{
          id: :swapper,
          start: {Swapper, :start_link, []}
        },
        %{
          id: :joiner,
          start: {Joiner, :start_link, []}
        }
      ]

      Supervisor.init(children, strategy: :one_for_all)
    end

    def process_message(message) do
```

```
      {_, splitter, _, _} = __MODULE__ |> Supervisor.which_children |> Enum.at(2)

      send(splitter, {message})
    end
  end
```

This code defines three processes (`Splitter` , `Swapper` , and `Joiner`) and a supervisor (`FormatterSupervisor`) that manages them. The purpose of this code is to split a string into words, swap the letter "m" with "n" in each word (case-insensitively), and then join the words back together into a new string.

The `Splitter` process receives a string, splits it into words, and sends the list of words to the `Swapper` process.

The `Swapper` process receives the list of words, transforms each word by swapping "m" with "n" (case-insensitively), and sends the transformed list of words to the `Joiner` process.

The `Joiner` process receives the list of transformed words, joins them together into a new string, and prints the new string to the console.

The `FormatterSupervisor` is responsible for starting the three processes and managing them. It provides a `process_message` function that can be used to send a message to the `Splitter` process.

Overall, this code demonstrates how processes can be used to split up a task into smaller, more manageable parts and to coordinate the execution of those parts.

Conclusion

While doing this project we got to learn some basic things about the actor model and functional programming

Bibliography

hexdocs.com stackoverflow.com https://elixir-lang.org/docs.html