**Terminal:**

* elixir –v = versión instalda
* elixir <nombre de archivo> para ejecutarlo
* iex.bat para ejecutar código de elixir “suelto”. (el .bat es en la term de VSCode).
  + Code.load\_file “<nombrearchivo>”: para ejecutar un archivo dentro de iex.
  + i <lo que sea>: es para inspeccionar el elemento, si es un integer, sus características, si puedo usar enum, etc.
  + c “nombre del archivo.ex”: compila el archivo. Una vez compilado puedo ejecutar cualquier función como NombreModulo.nombreFuncion(argumentos si existen)… siendo que el paréntesis es opcional. Para no tener que escribir mil veces el NombreModulo, puedo ejecutar “import NombreModulo”. Una vez importado, ya directamente puedo ejecutar “nombreFuncion” sin escribir previamente NombreModulo. Si cambio algo en el módulo, ejecuto “r <nombreModulo>” y con eso se hace un reload para tener las funciones con la última versión definida.
* IO.puts: imprime en terminal
* IO.gets: para que el usuario ponga un input, y se puede asignar a una variable. Ej, respuesta = IO.gets(“cómo te llamas?\n”)
* IO.inspect() = nos trae info del elemento a inspeccionar, por ej, el tipo de dato, una muestra de su valor, etc. Para ejecutarlo desde iex.bat se pone: i <elemento a inspeccionar>

**Condicionales:**

* case <evaluado> do

caso1 -> funcionX()

…

CasoN -> funcionZ()

\_ -> todo el resto de los casos

end

**Listas**

* [1, 2, 3] ++ [4,5] obtengo la unión de las dos listas
* Lists.first() me trae el primer elemento de la lista

**Tuples**

* Tupla = {“A”, “un string”, 1, 3.1416, :atom}. Puede contener cualquier tipo de dato. A diferencia de una lista, no se puede aplicar un Enum o reduce, etc, pero la ventaja es que se accede a cualquiera de sus elementos muy rápido ya que se guardan en memoria de manera continua, mientras que en una lista es muy “costoso” ya que no se guardan continuamente en memoria (si quiero acceder al elemento 8mil de la lista, tengo que pasar primero por los 7999 elementos).

**Atom**

* es una constante cuyo valor es su propio nombre. Son parecidos a los strings, pero con la característica de ser únicos, y hay un límite en la VM (máquina virtual) de la cantidad de atoms que simultáneamente podemos tener guardados. Unos cuántos ejemplos de atoms serían: :error, :ok, :paused …Como se puede ver en los ejemplos, un atom se define con : y unos cuantos caracteres, que representan su valor
* Por ejemplo, podemos tomar como norma, que el resultado devuelto en una operación de entrada salida, sea siempre una tupla. El primer valor de la tupla será un atom que representará el estado de la operación. En este caso usaremos :ok para las operaciones correctas y :error cuando se produce algún fallo. El segundo valor de la tupla será el resultado en el caso de que la operación sea correcta, o el mensaje de error en el caso contrario.

**Strings**

* “hola” <> “mundo” obtengo la unión “holamundo”
* ~s{}: para ingresar un string sin tener que hacerlo con “”. En lugar de poner “hola mundo”, puedo poner ~s{hola mundo}. Es útil cuando quiero poner un string que contenga “, ~{ingresar sólo “hola” o “mundo”}
* String.trim(): le saca, por ej, el \n
* String.split(): separa el string en una lista, según el separador elegido. Si no se especifica separa con “ “. Ej, string.split(“hola mundo”), el Split me traería [“hola”, “mundo”]. Con el segundo argumento puedo especificar el criterio para el Split, ej, separar según “99”. Ideal usarlo con regular expressions. Si como segundo elemento uso “”, me separa por ejemplo “hola” en [“h”, “o”, “l”,”a”, “”]. Para eliminar el último “” podría meter a continuación un |> Each.filter(fn x -> x != “” end).
* “””<párrafo> “”” para escribir un párrafo que después puedo imprimir con IO.puts
* String.downcase lo hace todo minúscula.

**Regular expressions**

* ~re/<expresión>/ o ~re{<expresión>}
  + \w: hace match con Word character,es decir, las letras, pero no los carácteres especiales como ().;|> etc.
  + [**^**<expresión>]: esto matchea todo lo que no sea la expresión.
  + [^\w]: match con todo menos los caracteres. Por ej, si lo meto como segundo argumento dentro de un Split, me va a separar por todo lo que no sea un carácter.
  + +: se usa para hacer match en uno o más de lo que definamos, si es que hay más de un match seguido. Por ej, si hago un Split de “hola()°|mundo” como [^\w]+ me va a hacer el Split cuando encuentre uno o más caracteres que no sean de letra. Como le puse el +, me considera el bloque “()°|” como uno sólo, y el resultado del Split será [“hola”, “mundo”]. Sin el más daría algo como [“hola”, “”, “”, “”,“”, “mundo”]
  + ~re{<match1>|<match2>}: el | se usa para que matchee **una u otra condición**. Si le agregara un + al final “(<match1>|<match2>)**+**”, eso haría match cuando encuentre uno o más <match1> o <match2> seguidos. Haría match con si el string aparece “…match1 match1 match2 match1…”.
  + [\\n](file:///\\n): si quiero que haga match con “\n” le agrego antes un \ adicional, ya que el \ se usa en re para poner el match deseado.
  + Dentro de un match puedo poner más de una cosa. Ej [^\w’] va a matchear con todo lo que no sea una letra (w) y con todo lo que no sea un ‘.

**Enum**

* .count(<lista o lo que sea a contar elementos>): cantidad de elementos contenidos.
* .filter(<condición de filtrado, una función anónima>): me devuelve la lista con los elementos que correspondan con el criterio para filtrar, que defino en general con una función anónima. Ej, <una lista cualquiera> |> Enum.filter(fn x -> x ! = “” end) me devuelve la lista con todos los elementos que sean distintos de vacío.
* .at(<enumerable>, <posición N>) me trae el elemento en la posición N del enumerable. Ej: Enum.at(lista, 3) me trae el tercer elemento de la lista en cuestión.
* .each(<enumerable>, <función a aplicar a cada enumerado del enumerable>). Ej, Enum.each(lista, fn elemento -> elemento\*2 end)
* .zip(<Enum1>, <Enum2>): combina dos Enums que son del mismo length y los mete dentro de una lista de tuplas. Ej, tengo una lista con los títulos = [nombre, apellido] y otra con los datos = [Nicolás, Cirio]. Como son del mismo lengthsi hago Enum.zip(títulos, datos) obtengo una lista con tuplas así: [{“nombre”, “Nicolás”}, {“apellido”, “Cirio”}], es decir, algo parecido a un map. Si quisiera que esto sea un map le aplico un Enum.into
* .into: Si al enum.zip(<enum1>, <enum2>) le hago un pipe a Enum.into(%{}) voy a obtener un map.
* .reduce(<enumerable>, <función reductora>) reduce, por ejemplo, los valores de una lista a un único valor. Es útil cuando quiero combinar varias cosas en una sola cosa. Ej, lista = [5, 7,9], y hago Enum.reduce(lista, fn a,b -> a + b end) el resultado será 21. Lo que hace es tomar primero a = 5 y b = 7, los suma y obtiene 12; 12 pasa a ser ahora el valor de a, y toma como b el siguiente valor de la lista, en este caso 9. Hace entonces ahora a = 12 + b = 9 y obtiene 21.

**Maps**

* Agrupación de “keys” con sus correspondientes “values”. Similar a los diccionarios de Python.
* map1 = %{:some\_key => “some value”, :otro\_key => 45, etc etc)
* map1.some\_key voy a obtener el valor correspondiente, en este caso, “some value”, siempre y cuando el key sea un atom. Si no fuera un atom, accedo al valor de la siguiente manera: map1[“key\_correspondiente”]
* Son **INMUTABLES,** es decir, una vez definidos no se pueden modificar en su extensión. Es decir, para agregar otro key-value pair tengo que crear otro map, con map.merge.
* map**.merge**(map1, %{:tercer\_key => 3,1416} y voy a obtener otro map así: %{:some\_key => “some value”, :otro\_key => 45, tercer\_key => 3,1416}.
* También existe map.put, investigarlo
* lo que sí se puede hacer es modificar el value de algún key que ya estaba en el map, de la siguiente manera: map1 = %{map1 | key\_ya\_existente: “nuevo valor para ese key”}. Notar que como para reasignar el valor del key puse **: después** del key. Cuando lo cree había puesto el : al ppio, ya que era un atom?
* Map**.keys** me trae un listado de todos los keys de ese map.

**Atributos**

@atributo del módulo: lo declaro al ppio se usa como una constante, ej: @autor “Nico”, y después poner en alguna función @autor para algo en particular

**Funciones**

* **defModule <nombre módulo> do … end:** un módulo que suele tener varias funciones adentro.
* **def <nombre función> (<argumentos opcionales>) <condiciones opcionales, ej, where…> do … end:** función “normal”.
* **<variable a asignar la función> = fn <argumentos> -> <output de la función> end :**  función anónima. Ej, duplicar = fn x -> x\*2 end. Esta función se la podría aplicar por ejemplo a todos los elementos de una lista o a un número. **Ej,** duplicar.(7) me va a dar 14. **Ej2:** no\_vacio = fn x -> x != “” end, va a chequear si el elemento que ingresa al a función no es vacío -> no\_vacio.(“hola”) va a dar true

**Pipeline**

**|> :** mete el resultado de la izquierda en la derecha, como primer argumento; si la función tuviera más argumentos, tendríamos que especificarlos. Ej:

name = IO.gets(“cuál es tu nombre?\n”) |> String.trim()

entonces el resultado el IO.gets lo mete en string lo “trimea”y ese resultado es el que termina guardando en la variable “name”.

Otra forma de hacerlo sería: name = String.trim IO.gets(…)

**Files**

* archivo = File.read(“ruta\_archivo”): abre el archivo y lo guarda en la variable ‘archivo’.
  + El retorno es {:ok, binary} donde binary es el contenido del archivo, o {:error, reason} donde reason es la razón que provocó el error
  + File.read! : con poner ! le decimos que si no lo puede abrir que tire error directamente

**Pattern matching**

* [a, b] = [1, 2] por lo tanto se da que a = 1 y que b = 2.
* Útil para separar una lista, por ej, [**head | tail** = lista. Por lo tanto en head queda el primer elemento de la lista, y en tail una lista con todos los demás elementos de la lista inicial. OBs. En lugar de head y tail podría haber escrito otra cosa, x ej, “encabezado” y “resto”. Otra opción para sería “hd <lista>” y obtengo el head de la lista, el primer elemento; y con “tl <lista>” obtendría los demás elementos
* Si hago “hd tl <lista>” obtengo el segundo elemento de la lista. Ya que el tail de la lista sería desde el segundo elemento en adelante, y el head de eso es el primer elemento.

**Elixir scripts (\*.exs)**

* Se ejecutan línea a línea, en lugar de ir llamando funciones… para ejecutar desde terminal escribir “elixir <nombre del archivo.ex**s**>”

**Shortcuts VSC**

* ctrl+alt+flecha abajo: copia debajo de donde estamos el bloque seleccionado. Si no seleccionamos nada, copia la fila justo abajo.
* Ctrl+d: selecciona la palabra donde estamos parados y resalta las palabras iguales. Si apretamos de nuevo ctrl+d nos selecciona además la próxima palabra igual, y así sucesivamente. Si escribimos después de haber seleccionado, nos cambia todas las palabra seleccionadas por lo que vayamos escribiendo.
* Ctrl+enter: me crea una nueva línea donde estoy y me pone el cursor en la nueva línea