

FCC Tool Kit

Generador de tablas de verdad

Operaciones entre conjuntos

Sucesiones

Relaciones y funciones



ITESO, Universidad
Jesuita de Guadalajara

Fundamentos de Ciencias Computacionales, primavera 2021

Profesor: Fernando Velasco Loera

Fecha de entrega: 13/05/2021

Equipo: M&M

Lilia Arceli Lobato Martínez, 706937

Laura Griselda González Camacho, 734049

FCC Tool Kit

Link de github: https://github.com/LiliaLobato/FCC_Practicas/blob/main/P5/FCCToolKit.py

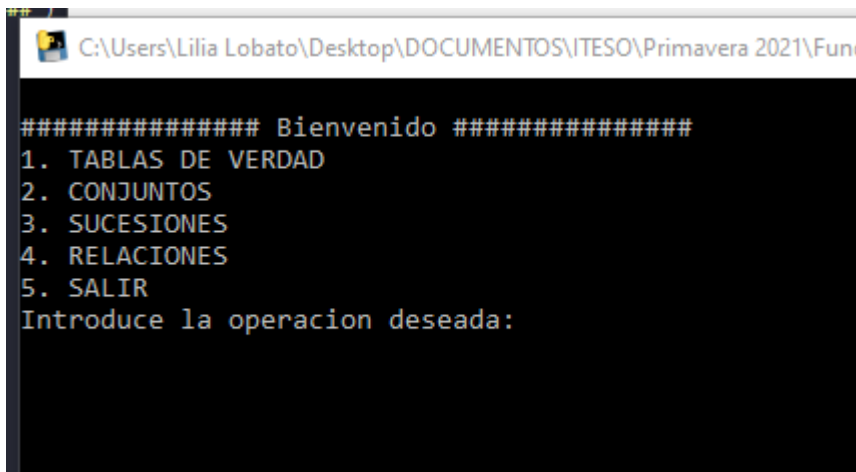
*El ejecutable se encuentra anexo al reporte

Funcionamiento a nivel usuario:

Al usuario se le presentarán las siguientes opciones:

- Generador de tablas de verdad
- Operaciones entre conjuntos
- Sucesiones
- Relaciones y funciones
- Salir

En caso de escoger alguna de las primeras cuatro opciones, la calculadora mostrará la interfaz respectiva en consola. Para saber de manera más detallada la interacción a nivel usuario que se debe de tener con la calculadora, a continuación se explican las diferentes modalidades. En caso de seleccionar la última opción, el proceso terminará. La siguiente imagen muestra el menú principal de la calculadora.



```
##### Bienvenido #####
1. TABLAS DE VERDAD
2. CONJUNTOS
3. SUCESIONES
4. RELACIONES
5. SALIR
Introduce la operacion deseada:
```

Documentación:

```
##### VARIABLES #####
salir = False
opcion = 0

##### MENU PRINCIPAL #####
while not salir:

    print("\n##### Bienvenido #####")
    print ("1. TABLAS DE VERDAD")
    print ("2. CONJUNTOS")
    print ("3. SUCESIONES")
    print ("4. RELACIONES")
    print ("5. SALIR")
```

```

opcion = int(input("Introduce la operacion deseada: "))

if opcion == 1: #TABLA DE VERDAD
    import truthTable
elif opcion == 2: #CONJUNTOS
    import conjuntos
elif opcion == 3: #SUCECIONES
    import sucesiones
elif opcion == 4: #RELACIONES
    import RelacionesFunciones
else:
    salir = True

print ("Gracias")

```

Esta parte del código despliega el menú principal y le permite al usuario seleccionar una opción. Dependiendo de la opción seleccionada, el código redirige al usuario al código de la operación que se desea realizar.

Generador de Tablas de Verdad

Link de github: https://github.com/LiliaLobato/FCC_Practicas/tree/main/P1

Funcionamiento a nivel usuario:

El generador de tablas de verdad permite al usuario ingresar una expresión en lenguaje lógico-matemático para que se le regrese la tabla de verdad por partes de dicha expresión. Los operadores lógicos que se pueden ingresar son:

- Conjunción, que se representa con un '&'.
- Disyunción, que se representa con un '|'.
- Negación, que se representa con '~'.
- Implicación, que se representa con '=>'.
- Bicondicional, que se representa con '<=>'.

Las proposiciones que el usuario puede ingresar son: p, q, r, s, t, u, v, w, x, y, z.

Lo único que deberá hacer el usuario es ingresar su expresión con los parámetros previamente indicados y recibirá su tabla de verdad, que imprimirá un 1 para referirse a Verdadero y un 0 para referirse a un Falso.

Ejemplo:

```

truthTable.py x *REPL* [python] x
#####
GENERADOR DE TABLAS DE VERDAD
#####
Para representar:
    AND usa &
    OR usa |
    NOT usa ~
    CONDICIONAL usa =>
    BICONDICIONAL usa <=>
Soporta las proposiciones: p q r s t u v w x y z
#####
Enunciados de logica proposicional: p&r

##### TABLA DE VERDAD #####
Enunciado: p&r

p r p&r
0 0 0
0 1 0
1 0 0
1 1 1

***Repl Closed***

```

Documentación:

```

def check_conditional(exp):
    if "=>" in exp:
        init= exp[exp.index("=")-1]
        final= exp[exp.index(">")+1]
        return "(~"+init+"|"+final+")"
    else:
        return exp

```

Verifica si el operador lógico del usuario es el de implicación (\Rightarrow). En caso de que sí sea, guarda la primera parte de la expresión en una variable, y la segunda parte de la expresión en otra. En el caso contrario, simplemente regresa `exp` que es la que recibe la función.

```

def space_print(exp, num):
    exp_sub = ""
    for i in range(num//2):
        exp_sub = exp_sub + " "
    exp_sub = exp_sub + exp
    for i in range(num//2):
        exp_sub = exp_sub + " "
    return exp_sub

```

El propósito de esta función es imprimir los espacios necesarios con la variable `exp` que recibe la función, la cantidad de veces que indica la variable `num` que también recibe la función.

```

print("#####\nGENERADOR DE TABLAS DE VERDAD\n#####")
print("Para representar:\n\t\t\t\t\tAND usa &\n\t\t\t\t\tOR usa |\n\t\t\t\t\tNOT usa ~")
print("\t\t\t\t\tCONDICIONAL usa =>\n\t\t\t\t\tBICONDICIONAL usa <=>")

```

```
print("Soporta las proposiciones: ", *suporter_var)
print("#####")
```

Esta parte imprime los parámetros para que el usuario los conozca.

```
input_exp = input("Enunciados de logica proposicional: ")
logic_exp = input_exp.lower()
logic_exp = logic_exp.replace(' ', '')
```

Esta parte del código se encarga de recibir la expresión del usuario, pasar la expresión a minúsculas y eliminar los espacios.

```
print("\n##### TABLA DE VERDAD #####")
print(f'Enunciado: {logic_exp}\n')
```

Aquí simplemente se regresa al usuario la expresión que ingresó en minúsculas y sin espacios.

```
name_regex = re.compile(r"[p-z]")
for i in range (len(suporter_var)):
    if suporter_var[i] in logic_exp:
        variables_list = variables_list + list(suporter_var[i])
```

Este for agrega las variables utilizadas en la expresión del usuario a *variables_list*.

```
res = re.findall(r'\(.*?\)', logic_exp)
```

Esta variable extrae los substrings de la expresión del usuario.

```
i = 0
for exp in res:
    #revisamos si la expresión contiene sub expresiones
    for letter in str(exp):
        if letter=="(":
            par_count+=1
    #si contiene subexpresiones
    if par_count > 1 :
        res_i = res[i]
        res[i] = res_i[1:]
        #las detectamos y ponemos como una expresion nueva
        sub_res = re.findall(r'\(.*?\)', res[i])
        res = res + list(sub_res)
    i+=1
    par_count = 0
res.append(str(logic_exp))
imp_res=res[:]
```

La primera parte de este for revisa si la expresión tiene paréntesis con un contador. Esto indicaría que se tiene que resolver primero la expresión dentro de los paréntesis. La segunda parte de este for verifica si el contador de paréntesis del for pasado es mayor a 1. Si es mayor a 1 se buscan las expresiones dentro de los

paréntesis y se almacenan dentro de otra variable. A la variable *res* se le agrega la variable *logic_exp*.

```
for exp in res:
    if "<=>" in exp:
        #encontramos las proposiciones
        init= exp[exp.index("<=>")-1]
        final= exp[exp.index("<=>")+3]
        #si son proposiciones compuestas, las identificamos
        if init=="(" :
            exp_sub = exp[:exp.index("<") ]
            exp_sub = exp_sub[exp_sub.index("("):exp_sub.index(")") +1]
            init = check_conditional(exp_sub)
        if final==")" :
            exp_sub = exp[exp.index("<=>")+3:]
            exp_sub = exp_sub[exp_sub.index("("):exp_sub.index(")") +1]
            final = check_conditional(exp_sub)
        #rearmamos la expresión en términos de AND, OR, NOT
        new_exp = "(~"+init+"|"+final+")&(~"+final+"|"+init+")"
        res[i] = new_exp
    i+=1
```

Esta parte revisa si es bicondicional. Primeramente guarda en dos variables distintas los caracteres a los lados del operador ' \leftrightarrow '. Después verifica si a los lados del operador se encuentran paréntesis. En caso de que sí sean paréntesis, guarda en otra variable la expresión que se encuentra en el paréntesis. Checa si la expresión es condicional con la función del principio. También guarda en *new_exp* el formato señalado en el código y eso lo almacena en la variable *res* en la posición *i*.

```
i = 0
for exp in res:
    res[i] = check_conditional(exp)
    i+=1
```

Este for únicamente recorre la expresión para verificar si es condicional con la ayuda de la función del principio.

```
number_rows = 2 ** len(variables_list)
for i in range(number_rows):
    ##Llenamos la tabla de verdad
    current_bin = bin(i)[2:].zfill(len(variables_list))
    for letter in str(current_bin):
        table_row = table_row + list(letter)
    table.append(table_row)

val_eval = table[i]
j = 0
```

Esta parte guarda el número de columnas necesarias en relación con el número de proposiciones que ingresa el usuario. Después comienza a llenar la tabla de verdad con la lista de letras.

```
for sub_exp in res:
    logic_exp = str(sub_exp)
    ## remplazamos cada variable por el valor de la columna
    for var in variables_list:
        logic_exp = logic_exp.replace(str(var), str(val_eval[j]))
        j += 1
    j=0

    ##Generamos el cambio hacia expresiones que python entiende
    logic_exp = logic_exp.replace("&", " and ")
    logic_exp = logic_exp.replace("|", " or ")
    logic_exp = logic_exp.replace("~", " not ")
    ##Se evalua esta expresión
    result = str(eval(logic_exp))
    result = "1" if (result=="True" or result == "1") else "0"
    ##Agregamos a nuestra tabla de verdad
    table[i] = table[i] + (list(result))
    logic_exp = sub_exp
    table_row = []
```

Esta parte evalúa las expresiones lógicas una por una y reemplaza las variables por el valor de la columna, reemplaza sus representaciones. Finalmente, asigna a la variable *result* que si es verdadero, será igual a 1, en el caso contrario será 0. La función *eval* de python toma una expresión en cadena y retorna un resultado en integer tras evaluarlo.

```
variables_list = variables_list + (imp_res)
print(*variables_list, sep = "\t")
#imprimimos la tabla y las evaluaciones
for row in table:
    tt=""
    i=0
    for item in row:
        space = len(str(variables_list[i]))
        tt=tt+space_print(item, space)+"\t"
        i+=1
    print(tt)
```

Esta es la parte final del código. Imprime la lista de variables separadas por una tabulación para la estructura de la tabla. Después asigna el valor correspondiente a la variable *tt* por cada parte de cada fila de la tabla y la imprime con una tabulación a un lado.

Operaciones entre conjuntos

Link de github: https://github.com/LiliaLobato/FCC_Practicas/tree/main/P2

Funcionamiento a nivel usuario:

La calculadora de teoría de conjuntos lee hasta 3 diferentes conjuntos ingresados por el usuario (separados por comas), y dependiendo de la operación que se desee hacer, regresa el nuevo conjunto. Las operaciones que hace son:

- Unión
- Intersección
- Diferencia
- Diferencia simétrica

La cardinalidad máxima de cada conjunto es de 10 elementos de tipo alfanumérico.

El usuario deberá comenzar por definir los elementos de los tres conjuntos, llamados 'A', 'B' y 'C' con los parámetros indicados anteriormente. Después, deberá



```
##### OPERACIONES DISPONIBLES #####
1. UNION
2. INTERSECCION
3. DIFERENCIA
4. DIFERENCIA SIMETRICA
5. SALIR
Introduce la operacion deseada: 1
1. A U B
2. B U C
3. A U C
4. A U (B U C)
Introduce los conjuntos a operar: 1
A U B: ['1', '2', '3', '4', '5'] U ['2', '3', '4', '5', '6', '7', '8']
['8', '6', '5', '3', '7', '2', '1', '4']
```

seleccionar cuál de las cuatro operaciones desea realizar, o bien, si desea terminar el programa. Una vez hecho esto, se desplegarán los diferentes acomodos para realizar la operación elegida y el usuario deberá seleccionar uno. El conjunto de respuesta se despliega en pantalla y el proceso se repite hasta que el usuario seleccione la opción de salir. En el siguiente ejemplo se muestra a un usuario seleccionando la operación de unión, y después seleccionando el primer conjunto.

Documentación:

UNIÓN:

```
def UNION_func(A,B):
    U = list(set(A) | set(B))
    return U
```

Esta es la función para la operación de unión. La unión son todos aquellos elementos que se encuentran en los conjuntos. En la función del código se usó la operación 'or' (|) para definir la variable 'U', que es el conjunto respuesta, y regresarla.

```
if opcion == 1:
    #UNION
```



```

print ("1. A U B")
print ("2. B U C")
print ("3. A U C")
print ("4. A U (B U C)")
sub_opcion = int(input("Introduce los conjuntos a operar:
"))
if sub_opcion == 1:
    print("A U B:" ,A," U ", B)
    print(UNION_func(A,B))
elif sub_opcion == 2:
    print("B U C:" ,B," U ", C)
    print(UNION_func(B,C))
elif sub_opcion == 3:
    print("A U C:" ,A," U ", C)
    print(UNION_func(A,C))
else :
    print("A U (B U C):" ,A," U ( ", B," U ",C," )")
    print(UNION_func(A,UNION_func(B,C)))

```

Aquí se le ofrecen al usuario los diferentes acomodos de los tres conjuntos para la operación Unión. Después, el código imprime la opción seleccionada y los conjuntos partícipes en la operación, e invoca la función explicada previamente con esos dos conjuntos.

Ejemplo:

```

##### OPERACIONES DISPONIBLES #####
1. UNION
2. INTERSECCION
3. DIFERENCIA
4. DIFERENCIA SIMETRICA
5. SALIR
Introduce la operacion deseada: 1
1. A U B
2. B U C
3. A U C
4. A U (B U C)
Introduce los conjuntos a operar: 1
A U B: ['1', '2', '3', '4', '5'] U ['2', '3', '4', '5', '6', '7', '8']
['8', '6', '5', '3', '7', '2', '1', '4']

```

INTERSECCIÓN:

```

def INTERSEC_func(A,B):
    N = list(set(A) & set(B))
    return N

```

Esta es la función para la operación de intersección. La intersección son solo aquellos elementos que se encuentran en ambos conjuntos. En la función del código se usó la operación 'and' ('&') para definir la variable 'N', que es el conjunto respuesta, y regresarla.

```

elif opcion == 2:
    #INTERSECCION
    print ("1. A n B")
    print ("2. B n C")
    print ("3. A n C")
    print ("4. A n (B n C)")
    sub_opcion = int(input("Introduce los conjuntos a operar:
"))
    if sub_opcion == 1:
        print("A n B:" ,A," n ", B)
        print(INTERSEC_func(A,B))
    elif sub_opcion == 2:
        print("B n C:" ,B," n ", C)
        print(INTERSEC_func(B,C))
    elif sub_opcion == 3:
        print("A n C:" ,A," n ", C)
        print(INTERSEC_func(A,C))
    else :
        print("A n (B n C):" ,A," n ( ", B," n ",C," )")
        print(INTERSEC_func(A,INTERSEC_func(B,C)))

```

Aquí se le ofrecen al usuario los diferentes acomodos de los tres conjuntos para la operación Intersección. Después, el código imprime la opción seleccionada y los conjuntos participantes en la operación, e invoca la función explicada previamente con esos dos conjuntos.

Ejemplo:

```

***** OPERACIONES DISPONIBLES *****
1. UNION
2. INTERSECCION
3. DIFERENCIA
4. DIFERENCIA SIMETRICA
5. SALIR
Introduce la operacion deseada: 2
1. A n B
2. B n C
3. A n C
4. A n (B n C)
Introduce los conjuntos a operar: 1
A n B: ['1', '2', '3', '4', '5'] n ['2', '3', '4', '5', '6', '7', '8']
['4', '5', '3', '2']

```

DIFERENCIA:

```

def DIFF_func(A,B):
    D = list(set(A)-set(B))
    return D

```

Esta es la función para la operación de diferencia. La diferencia son aquellos elementos que se encuentran en el primer conjunto que no se encuentran en el segundo. En la función del código se usó la operación de resta ('-') para definir la variable 'D', que es el conjunto respuesta, y regresarla.

```

elif opcion == 3:
    #DIFERENCIA
    print ("1. A - B")
    print ("2. A - C")
    print ("3. B - A")
    print ("4. B - C")
    print ("5. C - A")
    print ("6. C - B")
    sub_opcion = int(input("Introduce los conjuntos a
operar: "))
    if sub_opcion == 1:
        print("A - B:" ,A, " - ", B)
        print(DIFF_func(A,B))
    elif sub_opcion == 2:
        print("A - C:" ,A, " - ", C)
        print(DIFF_func(A,C))
    if sub_opcion == 3:
        print("B - A:" ,B, " - ", A)
        print(DIFF_func(B,A))
    elif sub_opcion == 4:
        print("B - C:" ,B, " - ", C)
        print(DIFF_func(B,C))
    elif sub_opcion == 5:
        print("C - A:" ,C, " - ", A)
        print(DIFF_func(C,A))
    else :
        print("C - B:" ,C, " - ", B)
        print(DIFF_func(C,B))

```

Aquí se le ofrecen al usuario los diferentes acomodos de los tres conjuntos para la operación Diferencia. Después, el código imprime la opción seleccionada y los conjuntos participantes en la operación, e invoca la función explicada previamente con esos dos conjuntos.

Ejemplo:

```

##### OPERACIONES DISPONIBLES #####
1. UNION
2. INTERSECCION
3. DIFERENCIA
4. DIFERENCIA SIMETRICA
5. SALIR
Introduce la operacion deseada: 3
1. A - B
2. A - C
3. B - A
4. B - C
5. C - A
6. C - B
Introduce los conjuntos a operar: 1
A - B: ['1', '2', '3', '4', '5'] - ['2', '3', '4', '5', '6', '7', '8']
['1']

```

DIFERENCIA SIMÉTRICA:

```
def DIFF_SIM_func(A,B):  
    D = list(list(set(A)-set(B)) + list(set(B)-set(A)))  
    return D
```

Esta es la función para la operación de diferencia simétrica. La diferencia simétrica son aquellos elementos que se encuentran en el primer conjunto que no se encuentran en el segundo, o que se encuentran en el segundo conjunto pero no se encuentran en el primero. En la función del código se usó la operación de resta ('-') entre los dos conjuntos y después la operación de suma ('+') entre ambas respuestas para definir la variable 'D', que es el conjunto respuesta, y regresarla.

```
elif opcion == 4:  
    #DIFERENCIA SIMETRICA  
    print ("1. A Δ B")  
    print ("2. A Δ C")  
    print ("3. B Δ A")  
    print ("4. B Δ C")  
    print ("5. C Δ A")  
    print ("6. C Δ B")  
    sub_opcion = int(input("Introduce los conjuntos a operar: "))  
    if sub_opcion == 1:  
        print("A - B:" ,A," - ", B)  
        print(DIFF_func(A,B))  
        print("B - A:" ,B," - ", A)  
        print(DIFF_func(B,A))  
        print("A Δ B:" ,A," Δ ", B)  
        print(DIFF_SIM_func(A,B))  
    elif sub_opcion == 2:  
        print("A - C:" ,A," - ", C)  
        print(DIFF_func(A,C))  
        print("C - A:" ,C," - ", A)  
        print(DIFF_func(C,A))  
        print("A Δ C:" ,A," Δ ", C)  
        print(DIFF_SIM_func(A,C))  
    if sub_opcion == 1:  
        print("B - A:" ,B," - ", A)  
        print(DIFF_func(B,A))  
        print("A - B:" ,A," - ", B)  
        print(DIFF_func(A,B))  
        print("B Δ A:" ,B," Δ ", A)  
        print(DIFF_SIM_func(B,A))  
    elif sub_opcion == 2:  
        print("B - C:" ,B," - ", C)  
        print(DIFF_func(B,C))  
        print("C - B:" ,C," - ", B)  
        print(DIFF_func(C,B))  
        print("B Δ C:" ,B," Δ ", C)  
        print(DIFF_SIM_func(B,C))  
    elif sub_opcion == 2:  
        print("C - A:" ,C," - ", A)  
        print(DIFF_func(C,A))  
        print("A - C:" ,A," - ", C)  
        print(DIFF_func(A,C))  
        print("C Δ A:" ,C," Δ ", A)
```

```

        print(DIFF_SIM_func(C,A))
    else :
        print("C - B:" ,C," - " , B)
        print(DIFF_func(C,B))
        print("B - C:" ,B," - " , C)
        print(DIFF_func(B,C))
        print("C Δ B:" ,C," Δ " , B)
        print(DIFF_SIM_func(C,B))

```

Aquí se le ofrecen al usuario los diferentes acomodos de los tres conjuntos para la operación Diferencia Simétrica. Después, el código imprime la opción seleccionada y los conjuntos participantes en la operación, e invoca la función explicada previamente con esos dos conjuntos.

Ejemplo:

```

##### OPERACIONES DISPONIBLES #####
1. UNION
2. INTERSECCION
3. DIFERENCIA
4. DIFERENCIA SIMETRICA
5. SALIR
Introduce la operacion deseada: 4
1. A Δ B
2. A Δ C
3. B Δ A
4. B Δ C
5. C Δ A
6. C Δ B
Introduce los conjuntos a operar: 1
A - B: ['1', '2', '3', '4', '5'] - ['2', '3', '4', '5', '6', '7', '8']
['1']
B - A: ['2', '3', '4', '5', '6', '7', '8'] - ['1', '2', '3', '4', '5']
['6', '7', '8']
A Δ B: ['1', '2', '3', '4', '5'] Δ ['2', '3', '4', '5', '6', '7', '8']
['1', '6', '7', '8']

```

Sucesiones

Link de github: https://github.com/LiliaLobato/FCC_Practicas/tree/main/P3

Funcionamiento a nivel usuario:

La calculadora de sucesiones le pide al usuario tres cosas:

- Fórmula en función de k
- Límite inferior (m)
- Límite superior (n)

La calculadora le regresa al usuario el desarrollo de la sucesión desde el límite inferior hasta el superior, indicando cuál era el valor de k en cada impresión. Realiza las operaciones básicas, necesarias para las sucesiones. Al final de la impresión del desarrollo de la sucesión, se imprime el cálculo de la sumatoria y de la multiplicación de todos los elementos entre los límites ingresados.

*Para calcular una potencia se debe poner de la siguiente manera: $k^2 = k**2$

En la siguiente imagen se muestra a un usuario ingresando la k como fórmula, con 1 y 5 como límites.

```
PROYECTO INTEGRAL FCC
TOOL KIT FCC
Sucesiones

Fórmula(k): k
Limite inferior: 1
Limite superior: 5

Desarrollo de la sucesión de m a n:
k:1 = 1
k:2 = 2
k:3 = 3
k:4 = 4
k:5 = 5

SUMATORIA: 15
MULTIPLICACIÓN: 120
```

Documentación:

El código de esta parte de la caja de herramientas fue desarrollado con la función de python `eval`. Después de pedir al usuario la fórmula y los límites, que guardamos en sus respectivas variables, realizamos un ciclo for que recorriera la variable local `i` desde el límite inferior m, hasta el límite superior n. Igualamos la variable k a i en y utilizamos la función `eval` para imprimir el desarrollo de la sucesión.

```
Formula = input("Fórmula(k): ")
m = int(input("Limite inferior: ")) #limite inferior
n = int(input("Limite superior: ")) #limite superior
print("")

suma = 0
mult = 1

print("Desarrollo de la sucesión de m a n:")

for i in range(m, n+1):
    k = i
    print("k:", i, " = ", round(eval(formula), 3), sep='')
    suma += eval(formula)
    mult *= eval(formula)
```

Para la sumatoria y la multiplicación inicializamos dos acumuladores antes del ciclo, el primero en 0 y el segundo en 1. Cada vuelta del ciclo, se le sumaba o multiplicaba respectivamente el resultado de la función *eval* con la fórmula dada por el usuario y el valor de *k* correspondiente. Finalmente, imprimimos la sumatoria y la multiplicación.

```
Print("\Nsumatoria:", suma)
print("MULTIPLICACIÓN:", mult)
```

Relaciones y Funciones

Link de github: https://github.com/LiliaLobato/FCC_Practicas/blob/main/P4

Funcionamiento a nivel usuario:

La calculadora de relaciones y funciones ayuda a determinar las siguientes características a la serie de parejas ordenadas ingresada por el usuario:

- Si tiene reflexividad
- Si tiene simetría
- Si tiene transitividad
- El dominio y el codominio
- Si es una función o no

El usuario debe ingresar primero los nodos que estarán presentes en su serie de parejas ordenadas. Después, deberá ingresar pareja por pareja. Cuando el usuario haya terminado de escribir las parejas, deberá escribir "done" para que la calculadora le devuelva los resultados. El código le regresa al usuario respuestas escritas acerca de las características indicadas en la lista previamente especificada. En el caso del dominio y del codominio, lo regresa en forma de lista.

En la siguiente imagen se muestra a un usuario ingresando las parejas ordenadas proporcionadas como ejemplo en el planteamiento de esta entrega del proyecto.

```

##### Bienvenido #####
1. TABLAS DE VERDAD
2. CONJUNTOS
3. SUCESIONES
4. RELACIONES
5. SALIR
Introduce la operacion deseada: 4

Nodos: 1,2,3,0

Relacion: 0,0
['0', '0']

Relacion: 0,1
['0', '1']

Relacion: 0,3
['0', '3']

Relacion: 1,0
['1', '0']

Relacion: 1,1
['1', '1']

Relacion: 2,2
['2', '2']

Relacion: 3,0
['3', '0']

Relacion: 3,3
['3', '3']

Relacion: done
['done']

##### RELACION #####
NODOS: ['1', '2', '3', '0']
RELACION: [('0', '0'), ('0', '1'), ('0', '3'), ('1', '0'), ('1', '1'), ('2', '2'), ('3', '0'), ('3', '3')]
##### ES REFLEXIVO #####
##### ES SIMETRICO #####
##### NO ES TRANSITIVO #####
DOMINIO: ['0', '1', '2', '3']
CODOMINIO: ['0', '1', '2', '3']
##### NO ES FUNCIÓN #####

```

Documentación:

```

arr = input("\nNodos: ") # Nodos
if (len(arr) != 0):
    Nodos = list(map(str, arr.split(','))) # quitamos las comas y
guardamos en un arreglo

Relacion_raw = "not done"
while Relacion_raw != ["done"]:
    Relacion_raw = input("\nRelacion: ") # Relacion
    if (len(Relacion_raw) != 0):
        Relacion_raw = list(map(str, Relacion_raw.split(','))) #
quitamos las comas y guardamos en un arreglo
        print(Relacion_raw)
        Relacion.append(tuple(Relacion_raw))
Relacion.pop(len(Relacion) - 1)

print("\n##### RELACION #####")
print("NODOS: ", Nodos)
print("RELACION: ", Relacion)

```



```
num_nodos = len(Nodos)
transiciones = len(Relacion)
```

En esta parte del código se solicita al usuario los nodos por ingresar, y se guardan a manera de array. Después, el código va guardando las parejas ordenadas ingresadas por el usuario en la variable *Relación*.

```
isReflexivo = 0
for i in range(transiciones):
    transicion_1 = (Relacion[i])
    node1_0 = transicion_1[0]
    node1_1 = transicion_1[1]
    if (node1_0 == node1_1): # validamos que sea un loop
        isReflexivo = isReflexivo + 1
    # print(transicion_1)

if (isReflexivo) >= num_nodos:
    print("##### ES REFLEXIVO #####")
else:
    print("##### NO ES REFLEXIVO #####")
```

Esta es la parte del código encargada de determinar si la relación es reflectiva o no. Hace esto por medio de un ciclo for que recorre el arreglo de parejas, e iguala las variables de modo que pueda comparar si el primer elemento de una pareja es igual al segundo.

```
isSimetrico = 0
for i in range(transiciones):
    transicion_1 = (Relacion[i])
    node1_0 = transicion_1[0]
    node1_1 = transicion_1[1]
    for j in range(transiciones):
        transicion_2 = (Relacion[j])
        node2_0 = transicion_2[0]
        node2_1 = transicion_2[1]
        if (node1_0 != node1_1): # validamos que no sea un loop
            if (node1_0 != node2_0) and (node1_1 != node2_1): # valida
que no sea el mismo
                if (node1_0 == node2_1) and (node1_1 == node2_0): #
valida que sea inverso
                    isSimetrico = isSimetrico + 1
                    # print(transicion_1,transicion_2)
            else:
                isSimetrico = isSimetrico + 1
                # print(transicion_1,transicion_2)

if isSimetrico % num_nodos == 0 and isSimetrico != 0:
    print("##### ES SIMETRICO #####")
else:
    print("##### NO ES SIMETRICO #####")
```

Esta parte del código revisa si la serie de parejas ordenadas es simétrica. Hace esto contando las veces que una pareja tiene el mismo elemento en ambas posiciones,

o las veces que un elemento encuentra su inversa dentro de la misma relación. Si al final el modulo de este contador y el número de nodos es igual a 0, la relación es simétrica. Si no es 0, no es simétrica.

```
isTransitivo = 0

for i in range(transiciones):
    transicion_1 = (Relacion[i])
    # obtenemos x,y
    node1_x = transicion_1[0]
    node1_y = transicion_1[1]
    for j in range(transiciones):
        transicion_2 = (Relacion[j])
        # obtenemos y,z
        node2_y = transicion_2[0]
        node1_z = transicion_2[1]

        if (node1_y == node2_y): # validamos que ambas Y sean las mismas
            node_y = node2_y

            for k in range(transiciones):
                transicion_3 = (Relacion[k])
                # obtngemos x,z
                node2_x = transicion_3[0]
                node2_z = transicion_3[1]
                if (node1_x == node2_x) and (node1_z == node2_z): #
Validamos que X y Z sean los mismos
                    node_x = node2_x
                    node_z = node2_z
                    if (str(node_x), str(node_z)) in Relacion:
                        print("Los nodos: x=", node_x, " y=", node_y, "
z=", node_z, "son transitivos")
                        isTransitivo = isTransitivo + 1
                    else:
                        isTransitivo = 0
                        print("Los nodos: x=", node_x, " y=", node_y, "
z=", node_z, "NO EXISTE")
                        # print(node2_x,node1_x)
                        # print(node2_y,node1_y)
                        # print(node2_z,node1_z)

if isTransitivo == transiciones:
    print("##### ES TRANSITIVO #####")
else:
    print("##### NO ES TRANSITIVO #####")
```

Esta parte del código se encarga de revisar si la relación es transitiva. Hace esto comparando los valores de la posición 'x', 'y' y 'z', que asigna con la ayuda de dos ciclos for. Si los valores de 'x' y 'y' de una pareja, y de 'y' y 'z' de otra existen en la relación, entonces los valores de 'x' y 'z' también deberían existir como pareja . Si es así, la relación es transitiva, de lo contrario no lo es.

```
##### CALCULAR DOMINIO
dominio = []
for i in range(transiciones):
    dominio.append(Relacion[i][0])

dominio = list(set(dominio))
dominio.sort()
print("DOMINIO: ", dominio)

##### CALCULAR CODOMINIO
codominio = []
for i in range(transiciones):
    codominio.append(Relacion[i][1])

codominio = list(set(codominio))
codominio.sort()
print("CODOMINIO: ", codominio)
```

Estas partes del código determinan el dominio y el contradominio de la relación. El dominio lo calcula guardando los primeros elementos de las parejas ordenadas en un nuevo arreglo, después los ordena con la función `.sort()` de Python y lo imprime. Lo mismo sucede para el contradominio, pero en lugar de guardar el primer elemento, guarda el segundo de las parejas ordenadas.

```
##### ES FUNCIÓN?
isFuncion = True;

for i in range (0, len(relacion)-1):
    for j in range (0, len(relacion)-1):
        if (i != j) and (relacion[i] == relacion[j]):
            isFuncion = True
        else:
            for k in range (0, len(relacion)):
                if (k != i):
                    if len(relacion[i]) == 1 or len(relacion[k]) == 1:
                        isFuncion = False
                    elif (relacion[i][0] == relacion[k][0]) and
(relacion[i][1] == relacion[k][1]):
                        isFuncion = True
                    elif relacion[i][0] == relacion[k][0]:
                        isFuncion = False

if isFuncion == True:
    print("##### ES FUNCIÓN #####")
else:
    print("##### NO ES FUNCIÓN #####")
```

Esta parte final del código determina si las parejas ordenadas pueden formar una función. Hace esto verificando que ninguna pareja tenga el mismo valor en 'x' que otra pareja. Si esta condición se cumple, sí es función. En el caso contrario no es función. La única excepción es si ambos elementos de la pareja se repiten en otra, no cuenta como romper la condición. Si hay un elemento en 'x' suelto, tampoco es función.

Relación de la materia con nuestras carreras

Los fundamentos de las ciencias computacionales se relacionan con nuestras carreras (Ingeniería en Sistemas Computacionales e Ingeniería en Electrónica) ya que al ser ingenierías, debemos conocer las bases de los pensamientos lógico-matemáticos. Los conocimientos adquiridos en este curso nos ayudarán a desarrollar nuestros proyectos de manera profesional, ya que estaremos mejor preparadas para dar solución a problemas de lógica de manera creativa y estructurada, con la ayuda de las proposiciones, las reglas de inferencia y las operaciones aprendidas. Los podremos utilizar para desarrollar códigos y circuitos con las operaciones básicas (and, or, etc.) con un mayor entendimiento de cómo funcionan y de los resultados que podemos esperar de estas.

Los temas vistos durante este curso nos ayudan a practicar la resolución de problemas de nuestra vida diaria también, con bases científicas y con una mejor organización. El conocer estos temas nos ayuda a comprender otros conceptos básicos para ambas carreras. La lógica proposicional, así como la teoría de conjuntos son las bases para comprender el funcionamiento de las herramientas que utilizamos para desarrollar proyectos en nuestros ámbitos de estudio. Para los sistemas computacionales, la lógica es la base de los lenguajes de programación y del desarrollo de códigos con funciones implementadas correctamente. Para la electrónica, constituyen la base de la implantación de circuitos por medio de lenguajes de descripción de hardware como Verilog y programación de FPGA. Finalmente, esta materia nos ha permitido ejercitar nuestras habilidades de programación, y también nos va a dejar con una serie de herramientas que nos serán útiles para materias y/o proyectos laborales futuros.

Referencias:

1. Lenka, C. (2018). *Python | Union of two or more lists*. Recuperado de: <https://www.geeksforgeeks.org/python-union-two-lists/>
2. Lenka, C. (2020). *Python | Difference between two lists*. Recuperado de: <https://www.geeksforgeeks.org/python-difference-two-lists/>
3. Lenka, C. (2018). *Python | Intersection of two lists*. Recuperado de: <https://www.geeksforgeeks.org/python-intersection-two-lists/>
4. SYMBOLAB (2021). *Calculadora de series*. Recuperado de: https://es.symbolab.com/solver/series-calculator/%5Csum_%7Bn%3D10%7D%5E%7B20%7D%20k%5E%7B2%7D