

FCC Tool Kit

**Generador de Tablas de Verdad**



ITESO, Universidad  
Jesuita de Guadalajara

Equipo: M&M

Lilia Arceli Lobato Martínez, 706937

Laura Griselda González Camacho, 734049

Link de github: [https://github.com/LiliaLobato/FCC\\_Practicas/tree/main/P1](https://github.com/LiliaLobato/FCC_Practicas/tree/main/P1)

### Funcionamiento a nivel usuario:

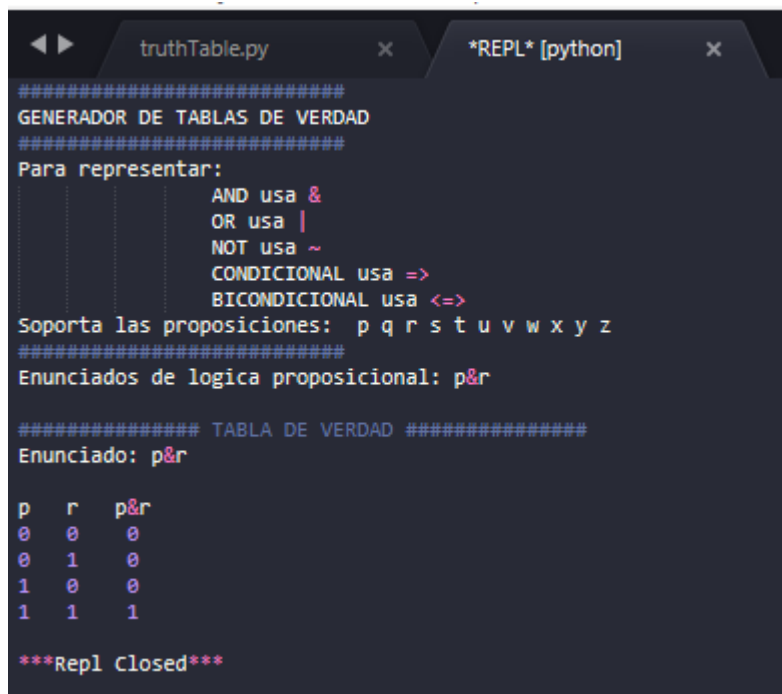
El generador de tablas de verdad permite al usuario ingresar una expresión en lenguaje lógico-matemático para que se le regrese la tabla de verdad por partes de dicha expresión. Los operadores lógicos que se pueden ingresar son:

- 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:



```
#####
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 ' $\lt;=>$ '. 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.