Inteligencia Artificial - IAI84

Instituto Tecnológico Metropolitano

Pedro Atencio Ortiz - 2018

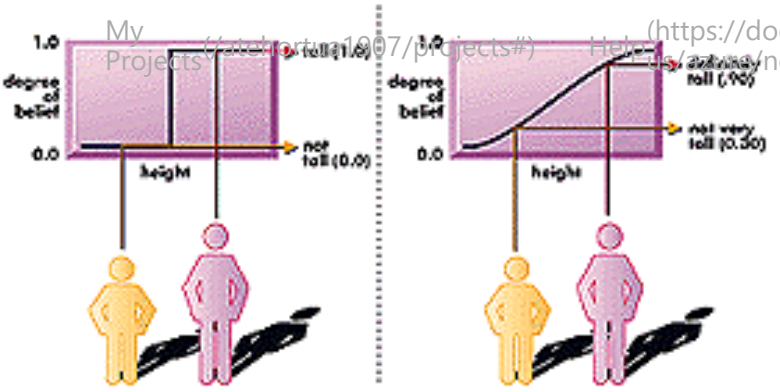
Parte de este material es autoría del profesor Cristian Guarnizo de la Universidad Tecnológica de Pereira. A él damos créditos.

En este notebook se aborda el tema de lógica difusa utilizando códigos ejecutables en Python:

- Conjuntos difusos.
- Funciones de pertenencia.
- Variable difusa.
- Fusificación (emborrosamiento).
- Base de reglas.
- Operadores difusos.
- Desfusificación.

Módulo 3_1: Logica Difusa

NON-FUZZY | FUZZY

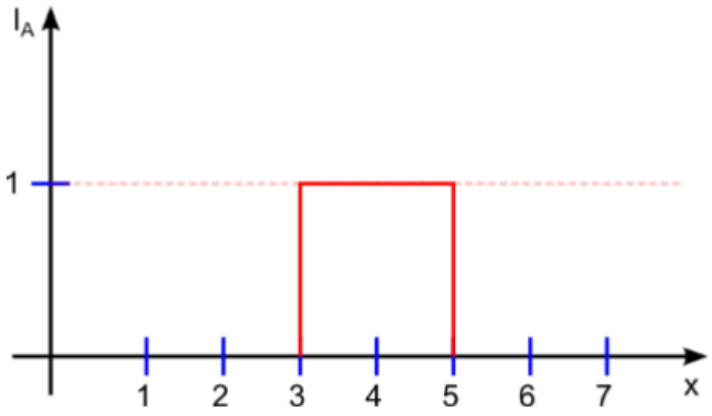


En términos reducidos, la lógica borrosa es un tipo de lógica que permite valores imprecisos (intermedios) para poder definir evaluaciones convencionales entre sí/no, verdadero/falso, negro/blanco, etc.

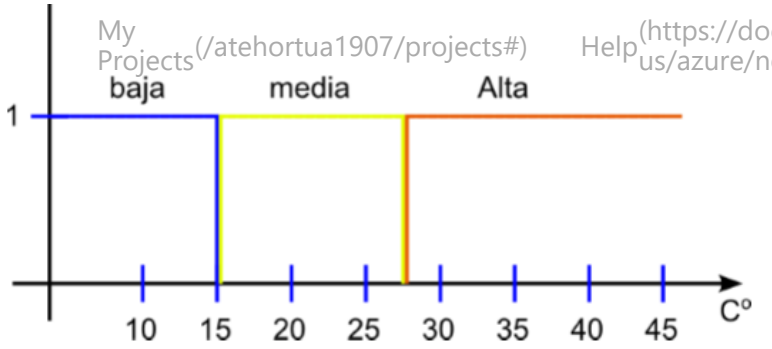
"El poder de la lógica difusa radica en que se puede analizar un sistema utilizando variables lingüísticas" [5].

Conjunto Booleano

En el algebra de Boole(George Boole) las variables toman valores falso/verdadero $(0, 1)$. Definamos un conjunto $A = [3, 5]$ y $x \in \{0, 7\}$. Los valores de x que pertenecen a A reciben valor verdadero (1).



Por ejemplo, para el caso de la variable temperatura, si quisiermos definir los valores temperatura baja, media y alta, utilizando lógica booleana, tendríamos:



En el ejemplo anterior, un elemento (valor de temperatura) tiene un grado de pertenencia a un conjunto o grupo, entonces, ¿A los 15 grados Celsius la temperatura es baja y repentinamente a los 16 es media?

In [10]:
```
"""
Implements a Boolean set.

Arguments:
x_domain -- domain of variable x (array)
x_set -- tuple of values that represents the set within domain of x

Returns:
parameters -- boolean (True, False) list of values inside x_set
"""
import numpy as np

def boolean_set(x_domain, x_set):
    val = np.logical_and(x_domain >= x_set[0], x_domain <= x_set[1])
    return val
```

Grafiquemos algunos conjuntos Booleanos para una variable x en el rango 0, 10

In [11]:
```
import matplotlib.pyplot as plt
```

```
limit1 = 0
limit2 = 100
domain = np.linspace(limit1, limit2, 500)

menor = (0.1,12)
bool_set_1 = boolean_set(domain,menor)

adolescente = (13,17)
bool_set_2 = boolean_set(domain,adolescente)

adulto = (18,90)
bool_set_3 = boolean_set(domain,adulto)

plt.fill(domain, bool_set_1)
plt.fill(domain, bool_set_2)
plt.fill(domain, bool_set_3)


plt.title("Three Boolean sets for variable x")
plt.show()
```

```
<Figure size 640x480 with 1 Axes>
```

In [12]:
```
import numpy as np

rango = np.linspace(0,20, 10)
print rango
```

```
[ 0.          2.22222222  4.44444444  6.66666667  8.88888889 11.111111
11
 13.33333333 15.55555556 17.77777778 20.        ]
```

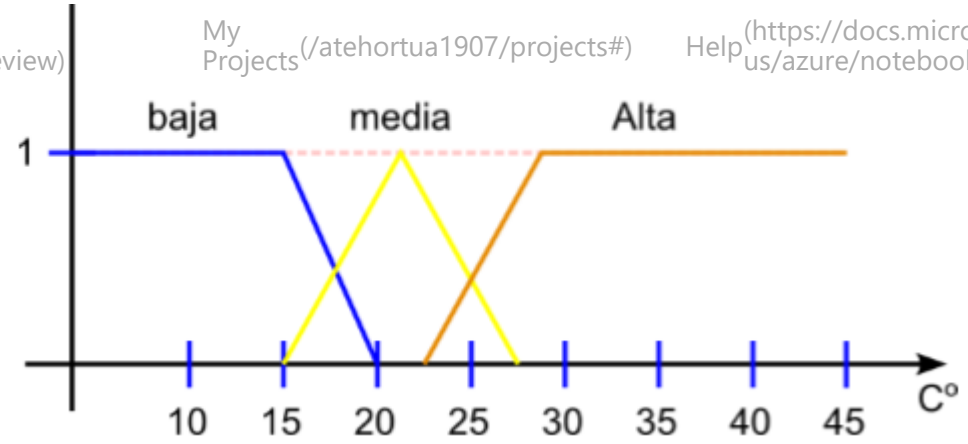
Trabajemos!

Implemente los conjuntos Booleanos que definen la mayoría de edad de una persona en Colombia.

Conjunto difuso (μ)

En un conjunto difuso, los grados de pertenencia son continuos y no binarios (falso, verdadero), de forma tal que una variable puede pertenecer en distinto grado a varios conjuntos.

En el siguiente ejemplo definimos tres conjuntos difusos μ_{baja}, μ_{media} y μ_{alta}, para la variable temperatura, en el rango [0, 45] grados centigrados.

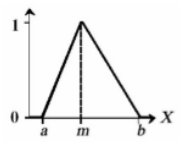


En el caso anterior, ¿un valor de temperatura igual a 17 grados será baja o media?

Funciones de pertenencia

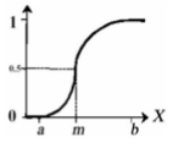
Un conjunto difuso μ se define mediante una función de pertenencia. Dicha función define la forma en que se distribuye la pertenencia de una variable x a un conjunto S cambia. A continuación, algunas funciones de pertenencia y su definición algebraica:

Triangular



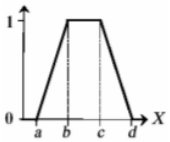
$$A(x)=\begin{cases} x\le a \rightarrow 0 \\ x\in[a,m] \rightarrow \dfrac{(x-a)}{(m-a)} \\ x\in(m,b) \rightarrow \dfrac{b-x}{b-m} \\ x\ge b \rightarrow 0 \end{cases}$$

Forma S



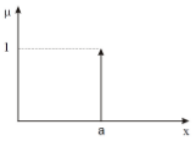
$$A(x)=\begin{cases} x\le a \rightarrow 0 \\ x\in[a,m] \rightarrow 2\left[\dfrac{(x-a)}{(b-a)}\right]^2 \\ x\in(m,b) \rightarrow 1-2\left[\dfrac{(x-a)}{(b-a)}\right]^2 \\ x\ge b \rightarrow 1 \end{cases}$$

Trapezoidal



$$A(x)=\begin{cases} (x\le a)\vee(x\ge d) \rightarrow 0 \\ x\in[a,b] \rightarrow \dfrac{(x-a)}{(b-a)} \\ x\in(b,c) \rightarrow 1 \\ x\in(b,d) \rightarrow \dfrac{d-x}{d-c} \end{cases}$$

Singleton



$$A(x)=\begin{cases} x=a \rightarrow 1 \\ x\ne a \rightarrow 0 \end{cases}$$

Dependiendo de la naturaleza de la variable que se estudia, se define el número de conjuntos difusos y la función de pertenencia de cada uno.

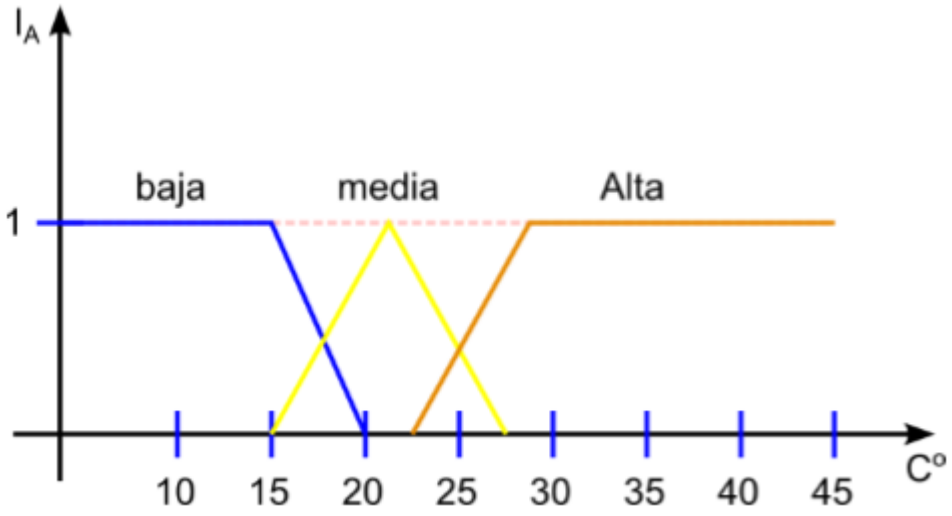
Trabajemos!

Defina los conjuntos difusos para la variable edad, respecto a las categorías infante, adolescente, adulto, cenil.

Variable difusa (x_{fuzzy})

Una variable difusa está conformada por los valores de pertencia de dicha variable respecto a cada conjunto. Entonces, la variable temperatura que estudiamos anteriormente se define como:

$$t_{fuzzy} = [\mu_{baja}(t), \mu_{media}(t), \mu_{alta}(t)]$$



Fusificación

Esta operación consiste en transformar una variable normal (crisp) en una variable fuzzy. Para ello, se aplica calcula por cada conjunto difuso, el grado de pertenencia de dicha variable. Por ejemplo para el caso de la temperatura, supongamos el caso $t = 10$, entonces:

- $\mu_{baja}(t) = 1$
- $\mu_{media}(t) = 0$
- $\mu_{alta}(t) = 0$

Por lo tanto $t_{fuzzy} = [1, 0, 0]$

In [13]:
```
diccionario = {"menor":17, "mayor":20}

print(diccionario["menor"])
```

```
17
```

In [14]:
```
"""
Implements a membership function
```

```
Implements a membership function.
Arguments:
x -- crisp variable
membership_function -- string with the name of membership function: "tra
parameters -- dictionary of parameters for membership function

Returns:
m_x -- float membership of x with respect to memebership_function
"""
import numpy as np

def membership(x, membership_function, parameters):
    m_x = 0

    if(membership_function == "trapezoid"):
        a = float(parameters["a"])
        b = float(parameters["b"])
        c = float(parameters["c"])
        d = float(parameters["d"])

        if(x <= a or x >= d):
            m_x = 0
        elif(x >= a and x <= b):
            m_x = (x-a)/(b-a)
        elif(x > b and x < c):
            m_x = 1
        elif(x >= c and x < d):
            m_x = (d-x)/(d-c)


    elif(membership_function == "triangular"):
        a = float(parameters["a"])
        m = float(parameters["m"])
        b = float(parameters["b"])

        if(x <= a):
            m_x = 0
        elif(x > a and x <= m):
            m_x = (x-a)/(m-a)
        elif(x > m and x < b):
            m_x = (b-x)/(b-m)
        else:
            m_x = 0

    return m_x
```

Trabajemos!

Utilizando las funciones de pertenencia implementadas, fusificar la variable $temperatura = 16.7 \in T$:

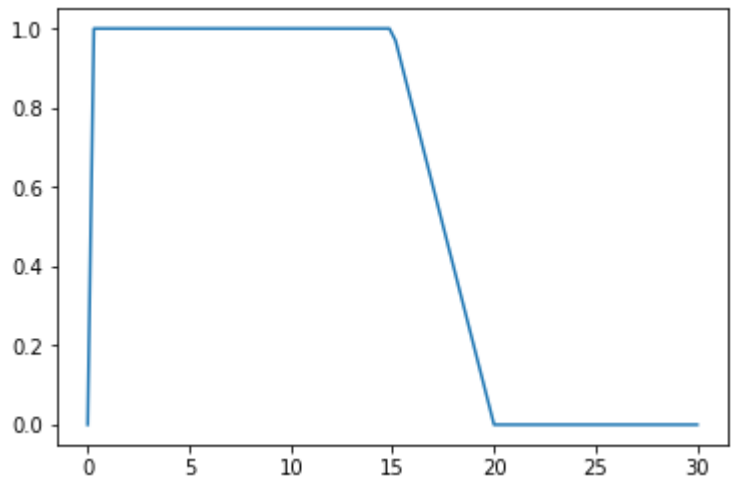
In [15]:
```
#Completar: aproximadamente 3 lineas de codigo
parameters_conjunto_baja = {"a":0, "b":0, "c":15, "d":20}
```

```
parameters_conjunto_media = {"a":15, "m":21.5, "b":27.5}
parameters_conjunto_alta = {"a":21.5, "b":27.5, "c":40, "d":40}

t = 16.7

#completar: aproximadamente 3 lineas de codigo
miu_baja = membership(t, "trapezoid", parameters_conjunto_baja)
miu_media = membership(t, "triangular", parameters_conjunto_media)
miu_alta = membership(t, "trapezoid", parameters_conjunto_alta)

#completar la lista de pertenencias
t_fuzzy = [miu_baja, miu_media, miu_alta]

print "temperatura fuzzy: ", t_fuzzy
```

```
temperatura fuzzy:  [0.6600000000000001, 0.26153846153846144, 0]
```

Obtengamos y visualicemos todo el conjunto fuzzy utilizando la función anterior

In [16]:
```
resolucion = 100
T = np.linspace(0, 30, resolucion)

#utilicemos comprehension de python para generar todo el vector del conj
conjunto_baja = np.array([membership(t, "trapezoid", parameters_conjuntc

plt.plot(T, conjunto_baja)
plt.show()
```



In [17]:
```
print T
print conjunto_baja
```

```
print conjunto_baja
```

```
[ 0.         0.3030303  0.60606061  0.90909091  1.21212121  1.515151
52
  1.81818182  2.12121212  2.42424242  2.72727273  3.03030303  3.333333
33
  3.63636364  3.93939394  4.24242424  4.54545455  4.84848485  5.151515
15
  5.45454545  5.75757576  6.06060606  6.36363636  6.66666667  6.969696
97
  7.27272727  7.57575758  7.87878788  8.18181818  8.48484848  8.787878
79
  9.09090909  9.39393939  9.6969697  10.         10.3030303  10.606060
61
 10.90909091 11.21212121 11.51515152 11.81818182 12.12121212 12.424242
42
 12.72727273 13.03030303 13.33333333 13.63636364 13.93939394 14.242424
24
 14.54545455 14.84848485 15.15151515 15.45454545 15.75757576 16.060606
06
 16.36363636 16.66666667 16.96969697 17.27272727 17.57575758 17.878787
88
 18.18181818 18.48484848 18.78787879 19.09090909 19.39393939 19.696969
7
 20.         20.3030303  20.60606061 20.90909091 21.21212121 21.515151
52
 21.81818182 22.12121212 22.42424242 22.72727273 23.03030303 23.333333
33
 23.63636364 23.93939394 24.24242424 24.54545455 24.84848485 25.151515
15
 25.45454545 25.75757576 26.06060606 26.36363636 26.66666667 26.969696
97
 27.27272727 27.57575758 27.87878788 28.18181818 28.48484848 28.787878
79
 29.09090909 29.39393939 29.6969697  30.        ]
[0.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         0.96969697 0.90909091 0.84848485 0.78787879
 0.72727273 0.66666667 0.60606061 0.54545455 0.48484848 0.42424242
 0.36363636 0.3030303  0.24242424 0.18181818 0.12121212 0.06060606
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.        ]
```

Creemos una función que permita generar el conjunto fuzzy de una variable X, dada la variable X, el tipo de conjunto fuzzy y sus parametros. **Esta función será de utilidad**

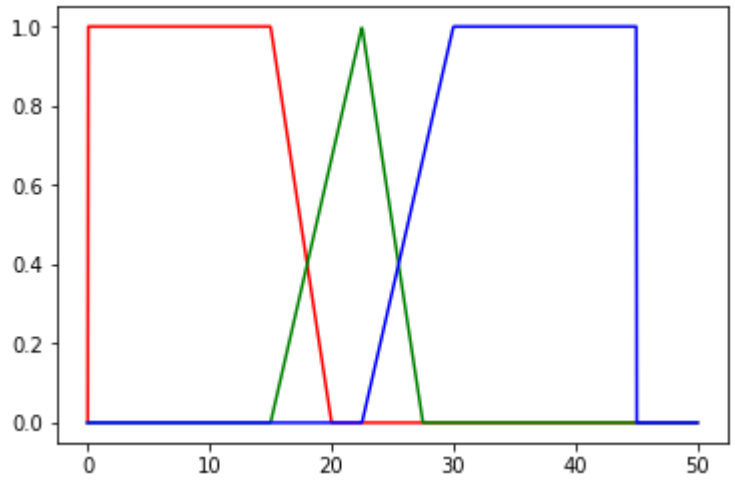
posteriormente.

In [80]:
```
def generate_fuzzy_set(X, membership_function, parameters):
    fuzzy_set = np.array([membership(x, membership_function, parameters)
    return fuzzy_set
```

Grafiquemos todos los conjuntos fuzzy de la variable T

Utilicemos para ello la funcion generate_fuzzy_set

In [19]:
```
params_conjunto_baja = {"a":0, "b":0, "c":15, "d":20}
params_conjunto_media = {"a":15, "m":22.5, "b":27.5}
params_conjunto_alta = {"a":22.5, "b":30, "c":45, "d":45}

resolucion = 1000
T = np.linspace(0, 50, resolucion)

conjunto_baja = generate_fuzzy_set(T, "trapezoid", params_conjunto_baja)
conjunto_media = generate_fuzzy_set(T, "triangular", params_conjunto_mec
conjunto_alta = generate_fuzzy_set(T, "trapezoid", params_conjunto_alta)

plt.plot(T,conjunto_baja, 'r')
plt.plot(T,conjunto_media, 'g')
plt.plot(T,conjunto_alta, 'b')
plt.show()
```



Trabajemos!

Implementar una función generica fusificacion que calcule automáticamente la fusificación de una variable x respecto a n conjuntos difusos:

In [20]:
```
"""
Implements a fuzzify function
```

```
Implements a fuzzify function.
Arguments:
x -- crisp variable
fuzzy_sets -- dictionary with next form: {"set_name":(membership_functic
    membership_function: string with the name of membership_function: "t
    parameters -- dictionary of parameters for membership function

Returns:
f_x -- dictionary of membership values of x with respect to each set in
"""
def fuzzify(x, fuzzy_sets):
    ##CODE HERE
    #f_x = []
    f_x = {} #lista de valores de pertenencia por cada conjunto difuso

    keys = fuzzy_sets.keys()

    for k in keys:
        #CODE HERE: utilizar la funcion membership(x, membership_functic
        #m = membership(x, fuzzy_sets[k]..., fuzzy_sets[k]...)
        m = membership(x, fuzzy_sets[k][0], fuzzy_sets[k][1])
        #f_x.append(k)
        f_x[k]=m

    return f_x
```

In [21]:
```
lista = [1, "hola", 3]
dic = {"1":1, "a":2}
tupla = (1,2)
```

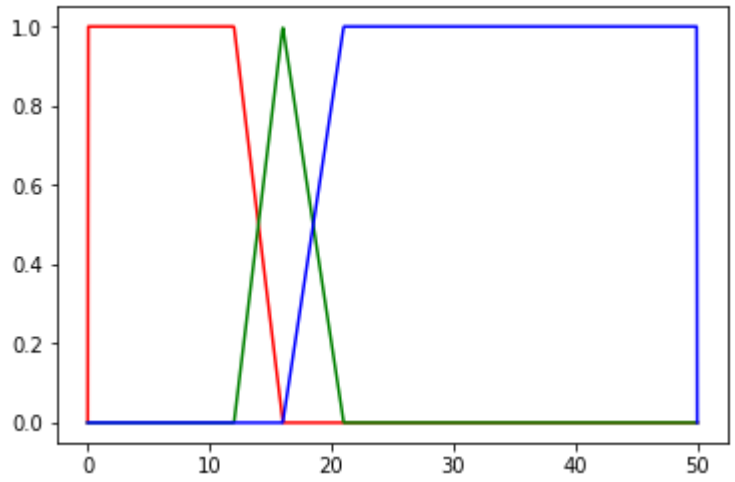
Utilicemos la función genérica para fusificar la variable t=16.7 $\in T$

In [22]:
```
params_conjunto_baja = {"a":0, "b":0, "c":15, "d":20}
params_conjunto_media = {"a":15, "m":22.5, "b":27.5}
params_conjunto_alta = {"a":22.5, "b":30, "c":45, "d":45}

funcion_baja = "trapezoid"
funcion_media = "triangular"
funcion_alta = "trapezoid"

key_baja = "baja"
key_media = "media"
key_alta = "alta"

t = 16.7
fuzzy_sets = {key_baja:(funcion_baja, params_conjunto_baja), key_media:(

print(fuzzify(t, fuzzy_sets))
```

```
{'baja': 0.6600000000000001, 'media': 0.22666666666666657, 'alta': 0}
```

In [23]:
```
EDAD = np.linspace(0,50, 1000)
edad = 13
```

```
edad = 13
params_conjunto_infante = {"a":0, "b":0, "c":12, "d":16}
params_conjunto_adolescente = {"a":12, "m":16, "b":21}
params_conjunto_adulto = {"a":16, "b":21, "c":50, "d":50}

funcion_infante = "trapezoid"
funcion_adolescente = "triangular"
funcion_adulto = "trapezoid"

f_set_infante = generate_fuzzy_set(EDAD, funcion_infante, params_conjunt
f_set_adolescente = generate_fuzzy_set(EDAD, funcion_adolescente, params
f_set_adulto = generate_fuzzy_set(EDAD, funcion_adulto, params_conjunto_

plt.plot(EDAD, f_set_infante, 'r')
plt.plot(EDAD, f_set_adolescente, 'g')
plt.plot(EDAD, f_set_adulto, 'b')
plt.show()

conjunto_infante = "infante"
conjunto_adolescente = "adolescente"
conjunto_adulto = "adulto"

diccionario_parametros = {conjunto_infante:(funcion_infante, params_conj


edad_fuzzy = fuzzify(edad, diccionario_parametros)

print("la variable concreta: ", edad, " fue fusificada como: ", edad_fuz
```




```
('la variable concreta: ', 13, ' fue fusificada como: ', {'adulto': 0,
'adolescente': 0.25, 'infante': 0.75})
```

In [24]:
```
edad_fuzzy["adolescente"]
```

Out[24]:
```
0.25
```

Salida esperada: [0.6600000000000001, 0.22666666666666657, 0]

Base de reglas

Estas reglas especifican el conocimiento del experto humano en forma de condicionales que utilizan los nombres de los conjuntos difusos que tienen las variables difusas. Es decir, estas reglas expresan la relación entre los valores lingüísticos de la entrada con respecto a la salida.

Por ejemplo, utilizando este tipo de reglas y las variables difusas, podemos realizar preguntas en lenguaje natural, tales como: ¿Está la temperatura alta?

Las reglas se construyen con instrucciones IF-THEN:

IF x_1 es "bajo" and x_2 es "medio" entonces y es medio

Estas reglas requieres de operadores AND y OR especificamente diseñados para la lógica difusa.

Operadores

Autor/Operador	A AND B	A OR B	NOT A
Zadeh	$min(\mu_A, \mu_B)$	$max(\mu_A, \mu_B)$	$1 - \mu_A$
Probabilístico	$\mu_A . \mu_B$	$\mu_A + \mu_B - \mu_A . \mu_B$	$1 - \mu_A$

Dependiendo de la definición de las reglas, los antecedentes se pueden combinar con los operadores AND y OR, o cambiar su valor con la negación NOT.

Resolución de reglas y obtención de la salida

La resolución de reglas se refiere a como se interpreta la regla desde la lógica difusa. El proceso es el siguiente:

1. Seleccionar la pertenencia a la cuál hacen referencia los operandos de las reglas:

$$\text{If } \mu_{bajo}(x_1) \text{ AND } \mu_{medio}(x_2) \text{ entonces } Y_{bajo}$$

2. Resolver los operadores difusos del antecendente:

$$K_{r1} = \text{AND}(\mu_{bajo}(x_1), \mu_{medio}(x_2))$$

3. Agrupar reglas con el mismo consecuente y aplicar operador OR:

$$A_{bajo} = \text{OR}(K_{r1} \in y_{bajo}, K_{r2} \in y_{bajo}, ..., K_{rn} \in y_{bajo})$$
$$A_{medio} = \text{OR}(K_{r1} \in y_{medio}, K_{r2} \in y_{medio}, ..., K_{rn} \in y_{medio})$$
$$A_{alto} = \text{OR}(K_{r1} \in y_{alto}, K_{r2} \in y_{alto}, ..., K_{rn} \in y_{alto})$$

4. Resolver el consecuente de la regla mediante implicación (MIN), para cada conjunto de la salida.

$$S_{bajo} = \text{MIN}(A_{bajo}, \mu_{bajo}(y))$$
$$S_{medio} = \text{MIN}(A_{medio}, \mu_{medio}(y))$$
$$S_{alto} = \text{MIN}(A_{alto}, \mu_{alto}(y))$$

5. Obtener el área de respuesta aplicando mediante AGREGACION utilizando el operador MAX entre todos los resultados de las reglas ya implicadas.

$$Y_{fuzzy} = MAX(S_{bajo}, S_{medio}, S_{alto})$$

6. Obtener el valor final de la regla mediante DESFUSIFICACION.

$$salida = desfusificacion(Y_{fuzzy})$$

---

2. Resolver operador difuso

In [25]:
```
"""
Implements Zadeh's Fuzzy Operators.

Arguments:
operator -- string with name of operator: "AND", "OR", "NOT"
parameters -- dictionary with parameters of operator

Returns:
k -- operators value
"""

def fuzzy_operator(operator, parameters):
    k = 0
    if(operator == "AND"):
        a = parameters["a"]
        b = parameters["b"]
        k = min(a, b)
    elif(operator == "OR"):
        a = parameters["a"]
        b = parameters["b"]
        k = max(a, b)
    elif(operator == "NOT"):
        a = parameters["a"]
        k = 1 - a
    else:
        print("Invalid operator.")

    return k
```

In [26]:
```
parameters = {"a": 0.6}
print fuzzy_operator("NOT", parameters)
```

```
print fuzzy_operator( 'NOT', parameters)
```

```
0.4
```

3. Agrupar reglas (mismo consecuente)

Reglas que tengan un mismo consecuente, son agrupas como un solo valor utilizando el operador OR == MAX (ZADEH), debido a que por lógica tradicional si dos antecedentes conducen a un mismo consecuente, es lo mismo que un disyunción entre dichos antecedentes, es decir:

$$\text{si } a < b \text{ entonces } c \text{ y si } a < f \text{ entonces } c$$

entonces, es lo mismo que:

$$\text{si } a < b \text{ OR } a < f \text{ entonces } c$$

In [27]:
```
"""
Implements common consecuent grouping.

Arguments:
rules -- list of values from resolved antecedents of rules with same cor

Returns:
val -- max from rules
"""

def fuzzy_group(rules):
    val = np.max(rules)
    return val
```

In [28]:
```
r1 = 0.2
r2 = 0.1
r3 = 0.6

rules = [r1,r2,r3]

print fuzzy_group(rules)
```

```
0.6
```

4. Resolver consecuente mediante implicación

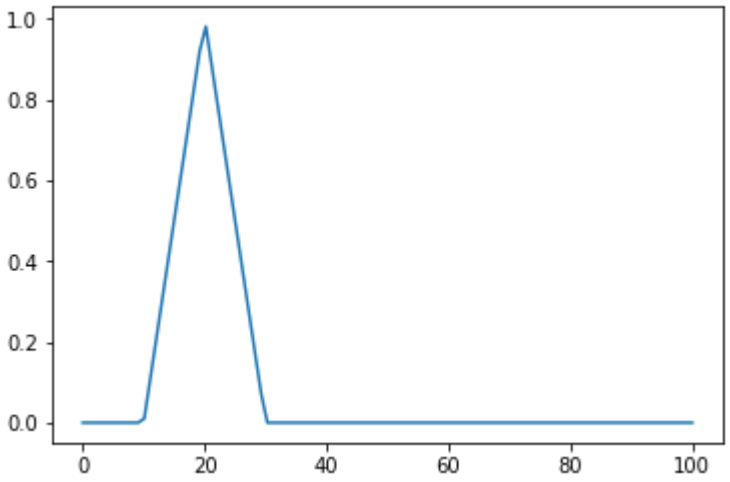
Lo primero que debemos hacer es generar el conjunto de salida al que hace referencia la regla o el conjunto de reglas ya agrupada.

Para ello utilizamos la funcion **generate_fuzzy_set** que construimos anteriormente.

supongamos un conjunto fuzzy A de una variable $Y \in [0, 100]$, de tipo triangular con parámetros $a = 10$, $m = 20$ y $c = 30$, realicemos una implicación sobre dicho

conjunto con valor de $r = 0.4$

In [29]:
```
resolucion = 100
Y = np.linspace(0,100,resolucion)
params_set_A = {"a":10, "m":20, "b":30}

fuzzy_set_A = generate_fuzzy_set(Y, "triangular", params_set_A)

plt.plot(Y, fuzzy_set_A)
plt.show()
```

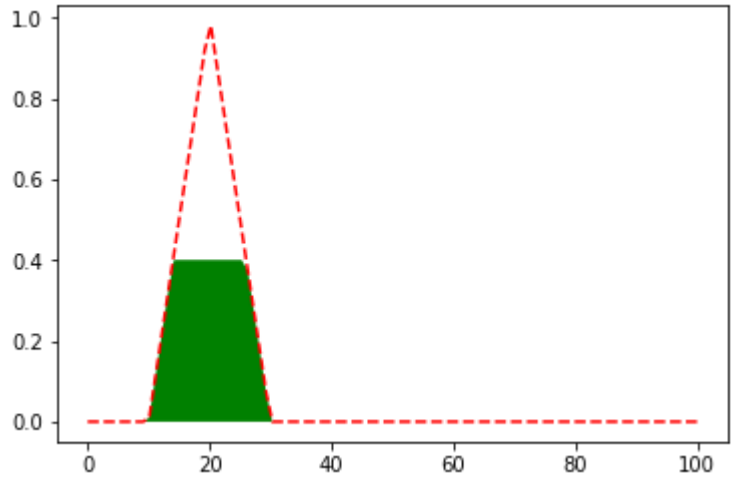


La implicación en lógica borrosa, se realiza mediante el operador **MIN**. Implementemos una función para ello:

In [30]:
```
"""
Implements fuzzy implication (MIN).

Arguments:
r -- scalar from rules solving and grouping.
fuzzy_set -- A fuzzy set (array of values).

Returns:
s -- implication result. An array of numeric values.
"""

def fuzzy_implication(r, fuzzy_set):
    val = np.minimum(r, fuzzy_set)
    return val
```

In [31]:
```
r = 0.4
implication_set = fuzzy_implication(r, fuzzy_set_A)
```

```
plt.plot(Y, fuzzy_set_A, 'r--')
plt.fill(Y, implication_set, 'g')
plt.show()
```

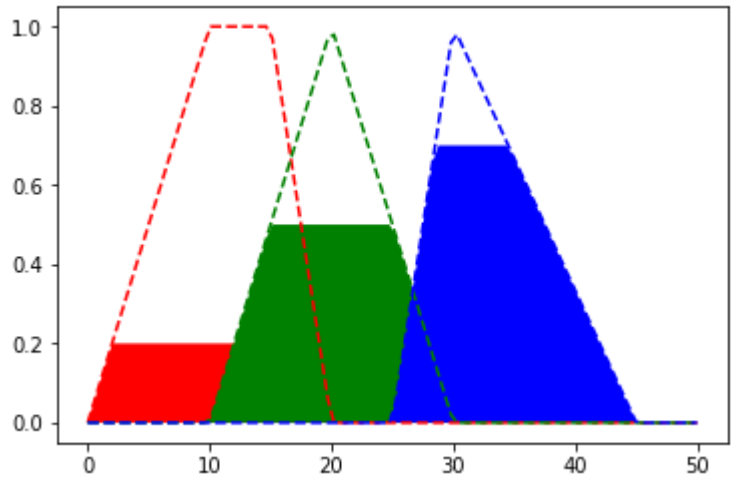


Supongamos que tenemos tres conjuntos: A de tipo trapezoidal, B de tipo triangular y C de tipo triangular para la variable de salida $Y \in [0, 50]$, y que hemos tenemos los valores de los antecedentes de tres reglas que hacen referencia a cada conjunto asi: r_A, r_B, r_C.

In [32]:
```
resolucion = 100
Y = np.linspace(0,50,resolucion)

params_set_A = {"a":0, "b":10, "c":15, "d":20}
params_set_B = {"a":10, "m":20, "b":30}
params_set_C = {"a":25, "m":30, "b":45}

Y_A = generate_fuzzy_set(Y, "trapezoid", params_set_A)
Y_B = generate_fuzzy_set(Y, "triangular", params_set_B)
Y_C = generate_fuzzy_set(Y, "triangular", params_set_C)

#1. apliquemos implicacion
r_A = 0.2
r_B = 0.5
r_C = 0.7

A_implicated = fuzzy_implication(r_A, Y_A)
B_implicated = fuzzy_implication(r_B, Y_B)
C_implicated = fuzzy_implication(r_C, Y_C)
```

In [33]:
```
plt.plot(Y, Y_A, 'r--')
```

```
plt.plot(Y, Y_C, 'b--')
plt.fill(Y, A_implicated, 'r')
plt.fill(Y, B_implicated, 'g')
plt.fill(Y, C_implicated, 'b')
plt.show()
```

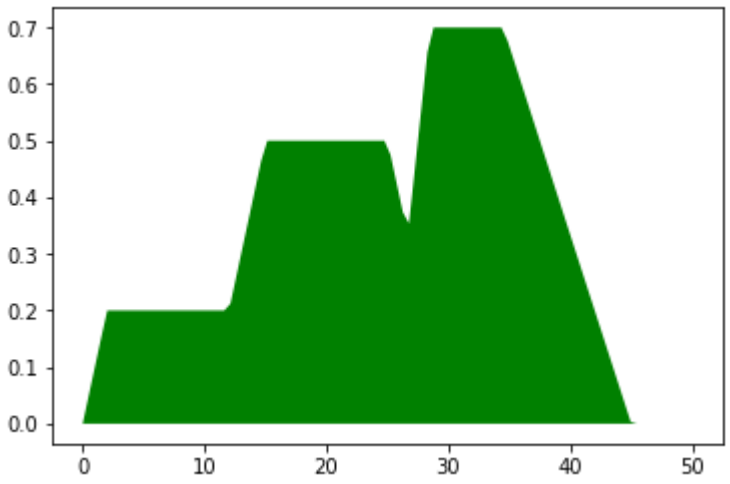


5. Obtener el área de respuesta aplicando mediante AGREGACION utilizando el operador MAX entre todos los resultados de las reglas ya implicadas

In [34]:
```
"""
Implements Aggregation fuzzy operator using MAX.

Arguments:
fuzzy_sets -- A list with fuzzy_sets of output variable. All sets must h

Returns:
val -- Aggregation result. An array of values.
"""
def fuzzy_aggregation(fuzzy_sets):
    val = np.zeros([fuzzy_sets[0].shape[0]])

    for s in fuzzy_sets:
        val = np.maximum(val, s)

    return val
```

In [35]:
```
agg_set = fuzzy_aggregation([A_implicated, B_implicated, C_implicated])
plt.fill(Y, agg_set, 'g')
```

```
plt.fill(i, agg_set, 'g')
plt.show()
```



In [36]:
```
print(agg_set)
```

```
[0.         0.05050505 0.1010101  0.15151515 0.2        0.2
 0.2        0.2        0.2        0.2        0.2        0.2
 0.2        0.2        0.2        0.2        0.2        0.2
 0.2        0.2        0.2        0.2        0.2        0.2
 0.21212121 0.26262626 0.31313131 0.36363636 0.41414141 0.46464646
 0.5        0.5        0.5        0.5        0.5        0.5
 0.5        0.5        0.5        0.5        0.5        0.5
 0.5        0.5        0.5        0.5        0.5        0.5
 0.5        0.5        0.47474747 0.42424242 0.37373737 0.35353535
 0.45454545 0.55555556 0.65656566 0.7        0.7        0.7
 0.7        0.7        0.7        0.7        0.7        0.7
 0.7        0.7        0.7        0.67676768 0.64309764 0.60942761
 0.57575758 0.54208754 0.50841751 0.47474747 0.44107744 0.40740741
 0.37373737 0.34006734 0.30639731 0.27272727 0.23905724 0.20538721
 0.17171717 0.13804714 0.1043771  0.07070707 0.03703704 0.003367
 0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.        ]
```

6. Obtener el valor final de la regla mediante DESFUSIFICACION

Para ello podemos aplicar diferentes métodos. El más conocido es el del CENTROIDE o centro de área, definido como:

$$y = \frac{\int_{i \in Y} i \mu_{aggregated}(i))di}{\int_{i \in Y} \mu_{aggregated}(i))di}$$

Otro método popular es el del BISECTOR definido como: "el valor que separa el área

bajo la curva en dos sub-áreas iguales":

$$y \mid \int_{i\in(-\infty,y)\in Y} i\mu_{aggregated}(i))di \approx \int_{i\in[y,\infty)\in Y} i\mu_{aggrega}$$

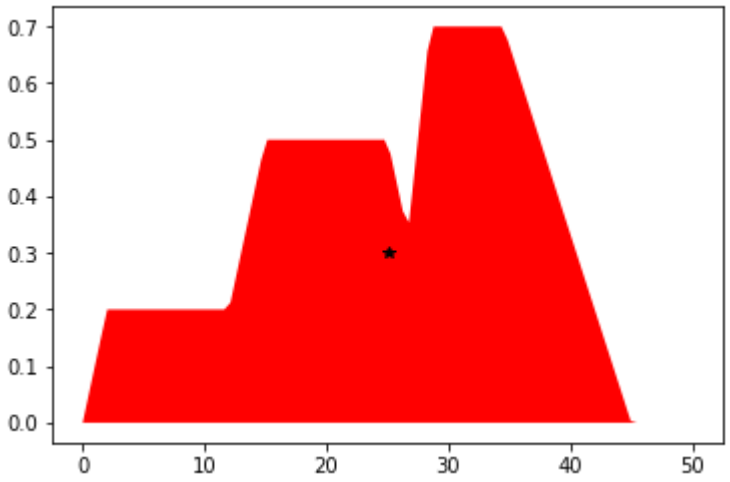
Implemetemos el método del centroide y apliquemos el mismo sobre el resultado anterior:

In [37]:
```
"""
Implements defuzzification (centroid, bisector).

Arguments:
Y -- array with range of output variable Y
fuzzy_set_output -- A fuzzy_set of output variable.
method -- string with name of deffuzification method. can be "centroid",

Returns:
val -- scalar value of crisp output variable.
"""
def fuzzy_defuzzy(Y, fuzzy_set_output, method="centroid"):

    if(method == "centroid"):
        val = np.sum(Y * fuzzy_set_output) /  np.sum(fuzzy_set_output)
    elif(method == "bisector"):
        val = None

    return val
```

In [38]:
```
output = fuzzy_defuzzy(Y, agg_set, "centroid")
print("salida del sistema: " , output)
```

```
agg_set = fuzzy_aggregation([A_implicated, B_implicated, C_implicated])
plt.fill(Y, agg_set, 'r')
plt.plot(output, 0.3, 'k*')
plt.show()
```

```
('salida del sistema: ', 25.03236833717073)
```



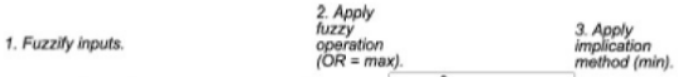
Taller en clase

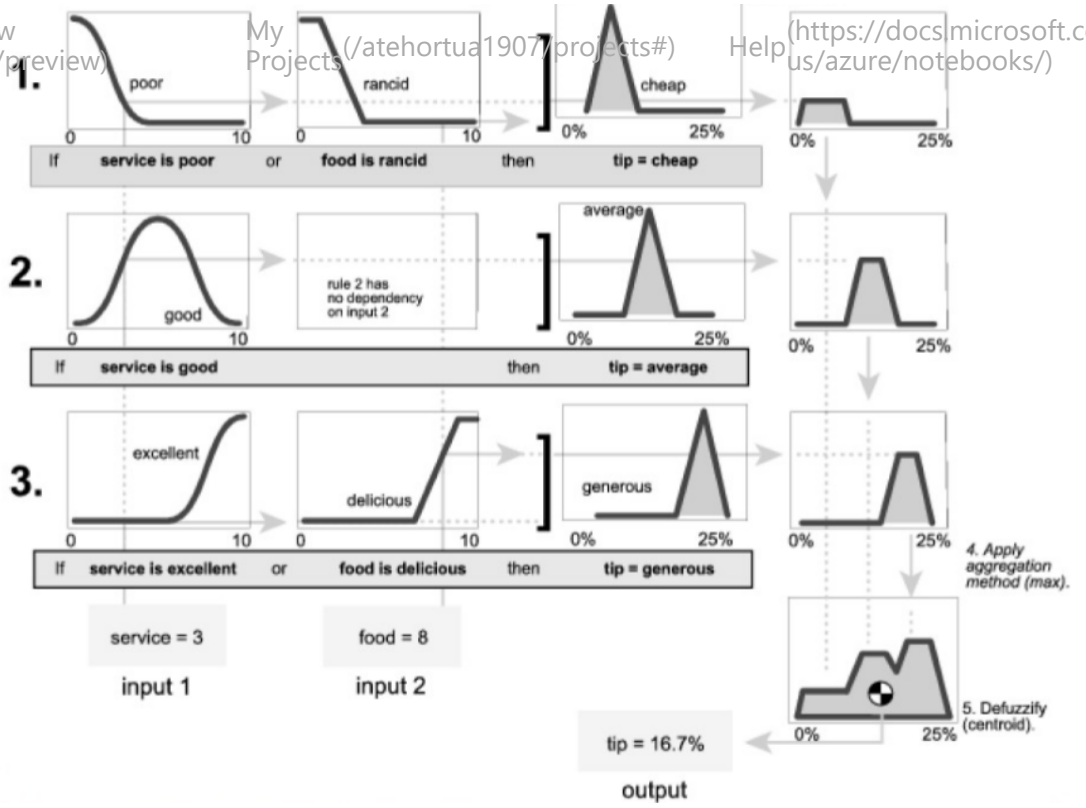
Recordermos! Hemos implementado las siguientes funciones:

- generate_fuzzy_set
- fuzzify
- fuzzy_operator
- fuzzy_group
- fuzzy_implication
- fuzzy_aggregation
- fuzzy_defuzzy

In [39]:
```
??generate_fuzzy_set
```

1. Implementemos el siguiente ejemplo y verifiquemos si nuestra implementación arroja un resultado similar:





In [82]:
```
#1. Inicializar parametros

service_poor = "poor"
service_good = "good"
service_excellent = "excellent"
funcion_service_poor = "trapezoid"
funcion_service_good = "triangular"
funcion_service_excellent = "trapezoid"
params_poor = {"a":0, "b":0, "c":2, "d":5}
params_good = {"a":2, "m":5, "b":8}
params_excellent = {"a":5, "b":7, "c":10, "d":10}


food_rancid = "rancid"
food_delicious = "delicious"
funcion_food_rancid = "trapezoid"
funcion_food_delicious = "trapezoid"
params_rancid = {"a":0, "b":4, "c":7, "d":4}
params_delicious = {"a":6, "b":8, "c":10, "d":10}

tip_cheap = "cheap"
tip_average = "average"
tip_generous = "generous"
funcion_tip_cheap = "triangular"
funcion_tip_average = "triangular"
funcion_tip_generous = "trapezoid"
params_cheap = {"a":0, "m":6, "b":12.5}
params_average = {"a":6, "m":12.5, "b":20}
params_generous = {"a":12.5, "b":20, "c":25, "d":30}
```

In [83]:
```
#2 fusificar
```

```
service = 9
food = 9

fuzzy_sets_servicio = {service_poor:(funcion_service_poor, params_poor),
service_fuzzy = fuzzify(service, fuzzy_sets_servicio)

fuzzy_sets_food = {food_rancid:(funcion_food_rancid, params_rancid), foo
food_fuzzy = fuzzify(food, fuzzy_sets_food)

print(service_fuzzy)
print(food_fuzzy)
```

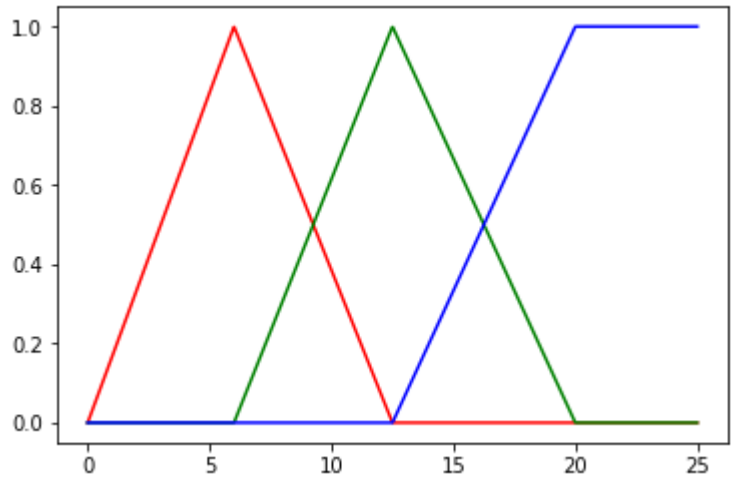
```
{'poor': 0, 'good': 0, 'excellent': 1}
{'delicious': 1, 'rancid': 0}
```

In [71]:
```
??generate_fuzzy_set
```

In [84]:
```
#3. Resolver reglas
r1 = fuzzy_operator("OR", {"a":service_fuzzy["poor"], "b":food_fuzzy["ra
r2 = service_fuzzy["good"]
r3 = fuzzy_operator("OR", {"a":service_fuzzy["excellent"], "b":food_fuzz

print(r1,r2,r3)
```

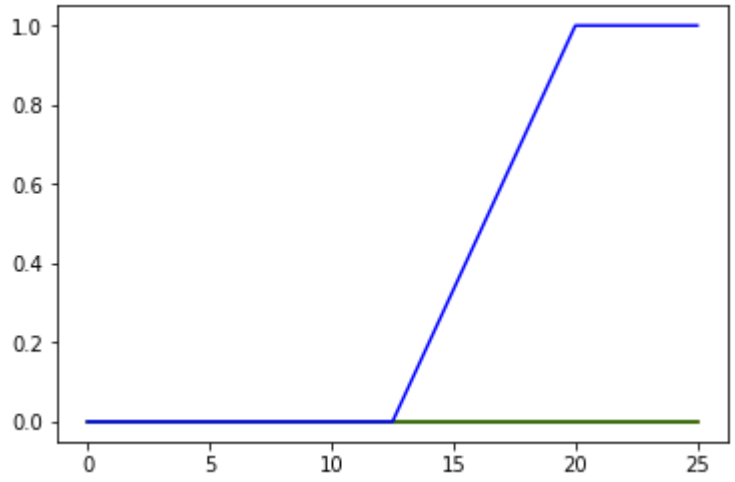
```
(0, 0, 1)
```

In [85]:
```
#4. Implicacion
T = np.linspace(0,25, 1000)
fuzzy_set_cheap = generate_fuzzy_set(T, funcion_tip_cheap, params_cheap)
fuzzy_set_average = generate_fuzzy_set(T, funcion_tip_average, params_av
fuzzy_set_generous = generate_fuzzy_set(T, funcion_tip_generous, params_

plt.plot(T, fuzzy_set_cheap, 'r')
plt.plot(T, fuzzy_set_average, 'g')
plt.plot(T, fuzzy_set_generous, 'b')
plt.show()
```



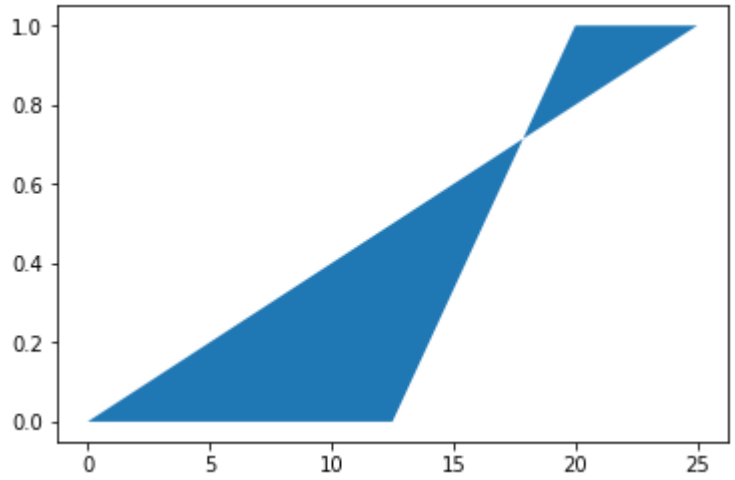
In [45]:
```
??fuzzy_implication
```

In [86]:
```
fs_cheap_truncated = fuzzy_implication(r1, fuzzy_set_cheap)
fs_average_truncated= fuzzy_implication(r2, fuzzy_set_average)
fs_generous_truncated = fuzzy_implication(r3, fuzzy_set_generous)

plt.plot(T, fs_cheap_truncated, 'r')
plt.plot(T, fs_average_truncated, 'g')
plt.plot(T, fs_generous_truncated, 'b')
plt.show()
```



In [47]:
```
??fuzzy_aggregation
```

In [87]:
```
#5. Agregacion
aggregated_tip = fuzzy_aggregation([fs_cheap_truncated, fs_average_trunc
plt.fill(T, aggregated_tip)
plt.show()
```



In [55]:
```
??fuzzy_defuzzy
```

In [88]:
```
#6. desfusificar
salida = fuzzy_defuzzy(T, aggregated_tip, "centroid")
```

```
print("al señor se le recomienda pagar: ", salida)
```

```
('al se\xc3\xb1or se le recomienda pagar: ', 20.3637754869411)
```

In [ ]: