Abstracciones

Apuntes versión 1.0

Víctor Araya Sánchez

2017

Objetivos de este apunte

Al terminar de leer, hacer los ejercicios propuestos de este apunte y trabajar en clases serás capaz de:

- - Diseñar abstracciones de problemas acotados de distinta índole, considerando las buenas prácticas asociadas.
- - Implementar abstracciones a través del lenguaje de programación Python, considerando las buenas prácticas de programación.

1. Introducción

El presente apunte tiene como fin que aprendas a realizar abstracciones de distintos tipos de problemas, para luego implementarlas en código funcional.

Nuevamente partiremos por lo básico… ¿Qué es una abstracción?

abstraer Conjugar

Del lat. *abstrahĕre.*
Conjug. c. *traer.*

1. tr. Separar por medio de una operación intelectual un rasgo o una cualidad de algo para analizarlos aisladamente o considerarlos en su pura esencia o noción. U. t. c. intr. *Pensar es olvidar diferencias, es generalizar, abstraer.*

2. intr. Hacer caso omiso de algo, o dejarlo a un lado. *Centremos la atención en lo esencial abstrayendo* DE *consideraciones marginales.* U. t. c. prnl.

3. prnl. Concentrarse en los propios pensamientos apartando los sentidos o la mente de la realidad inmediata. *Aspirando suavemente su cigarro, se abstrajo* DEL *ruido de la disputa.*

4. prnl. Retirarse o recogerse, apartándose del trato social. *Abstraerse* DE *los negocios mundanos.*

En este caso nos fijaremos en la primera acepción, entendiendo la abstracción con el efecto o consecuencia de abstraer, que aparecen en el cuadro anterior: Entenderemos una abstracción como el análisis de un problema y su separación en distintas partes, que llamaremos subproblemas, con foco, para nuestro caso, en su resolución.

En estricto rigor una abstracción es independiente de cualquier lenguaje de programación, por lo que una misma abstracción podría implementarse en Python, como en C++, como en otros posibles lenguajes. No obstante, conocer las potencialidades del lenguaje con el que se trabaja permite tener algunas ideas a priori que podrían resultar, a veces, de utilidad, y otras veces como obstáculos.

Pero, ¿por qué preocuparnos de la abstracción?

Cada vez que uno se enfrenta a un problema, que requiere de la elaboración de un programa para su solución, es natural sentir la tentación imperiosa de programarla inmediatamente: algún `while` por aquí, otro `if` por allá... y, eventualmente, el problema se resolverá, en el mejor de los casos, más por acumulación de código que por saber qué está ocurriendo, o caeremos en la frustración de no resolver el problema, ni entender lo que escribimos como código.

Para evitar la situación descrita anteriormente es que hacemos énfasis en la abstracción: en la medida que los problemas y programas se van volviendo más complejos, se requiere invertir más tiempo en diseñar una solución que en implementarla. A través de esta técnica buscamos identificar subprocesos que son necesarios de aplicar para generar el resultado final.

2. Diseño de abstracciones

Es importante comenzar explicitando que a lo largo de este tema trabajaremos considerando la estructura de buenas prácticas que se ha discutido anteriormente en el curso, teniendo en mente siempre las entradas, salidas y procedimientos de cada problema y subproblema que encontremos.

Para realizar el procedimiento de abstracción utilizaremos un problema concreto que iremos analizando paso a paso.

Ejemplo

Se requiere de un programa desarrollado en Python que encuentre puntos del plano cartesiano que cumplan la condición de pertenecer a una elipse, considerando su definición como lugar geométrico.

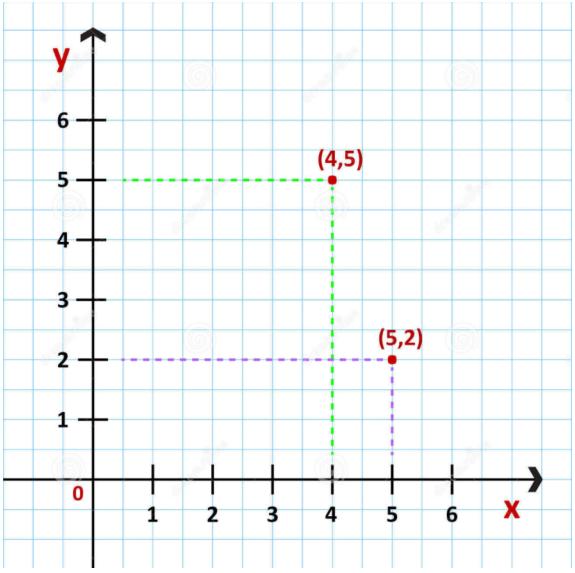
Siguiendo la estructura descrita lo primero que debes haces es identificar qué sabes del problema y qué no sabes, con el fin de investigarlo y complementar el problema con dicha información.

Algunas preguntas que podrían salir del problema planteado son:

- ¿Qué es un punto?
- ¿Qué es un plano cartesiano?
- ¿Qué es una elipse?
- ¿Qué es un lugar geométrico?
- ¿Cómo se describe la elipse en términos de lugar geométrico?

Es posible que conozcas las respuestas de algunas de ellas, pero precisaremos en 3 que son de vital importancia para entender el problema:

1. Que se solicite que los puntos pertenezcan a un plano cartesiano nos dice que son puntos que podemos describir como pares ordenados de la forma (x,y), donde x representa la posición del punto en el eje de las abscisas e y representa la posición en el eje de las ordenadas.



2. Un lugar geométrico corresponde a un conjunto de puntos que cumplen una determinada condición.
3. Una elipse es el lugar geométrico de todos los puntos de un plano, tales que la suma de las distancias a otros dos puntos fijos, llamados focos, es constante.

Teniendo un poco más claros los elementos constitutivos del problema debemos responder:

- ¿Cuáles son las entradas del problema?
- ¿Cuáles son las salidas del problema?

Cuando el problema no es explícito en definir las entradas y salidas con las que debe trabajarse se hace necesario tomar algunas decisiones que permitan solucionarlo. En este caso optaremos por definir entradas y salidas como sigue:

- Entradas:
  - Focos de la elipse (porque son necesarias por su definición como lugar geométrico) como puntos en forma de pares ordenados (dado que trabajaremos en un plano cartesiano)
  - Un valor numérico que determine el valor constante equivalente a la suma de las distancias desde los focos de la elipse a los puntos de esta (por la definición de elipse como lugar geométrico). Este valor será entero (decisión arbitraria, a veces pueden tomarse estas decisiones para probar soluciones más simples y luego se va variando para hacer soluciones más completas).
  - Un valor numérico que determine la cantidad de puntos a encontrar (por lo que solicita el problema, dado que en lo práctico no es posible encontrar a todos los puntos de una elipse, que son infinitos). Este valor debe ser entero.
- Salidas:
  - Tantos puntos como los solicitados que pertenecen a la elipse en forma de pares ordenados (por estar trabajando en un plano cartesiano), agrupados en una lista (Para organizar la información).

Actividad

¿Por qué el valor que determina la cantidad de puntos a encontrar debe ser entero?

A continuación, es necesario que nos cuestionemos cómo lograr solucionar el problema, considerando las entradas disponibles y las salidas declaradas.

Actividad

Dedica 10 minutos a pensar ¿cómo resolverías el problema? Anota tus ideas y tenlas en mente para lo que sigue

Para solucionar el problema tendremos en mente la siguiente pregunta:

¿Cuáles son los procesos o tareas que tengo que realizar para generar la salida a partir de la entrada?

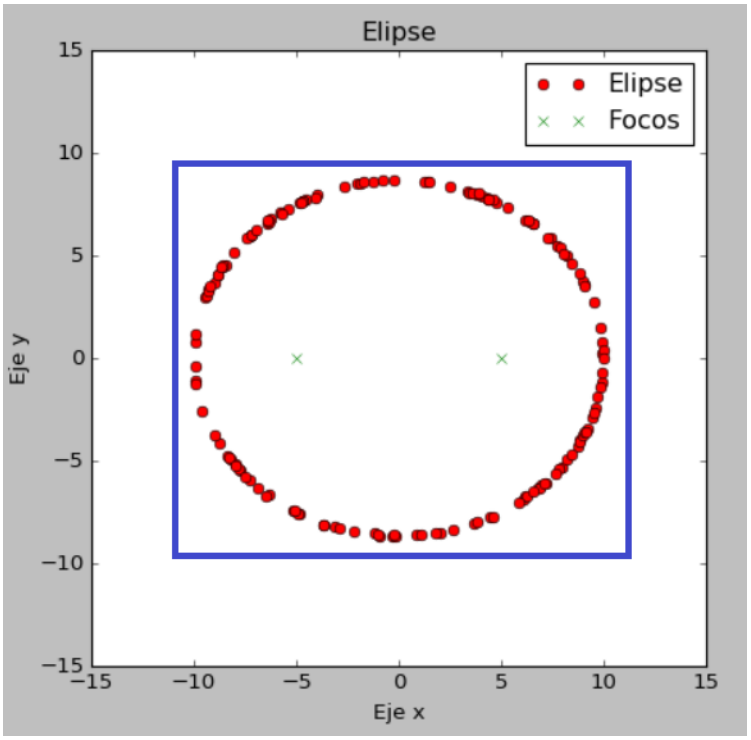
Para el problema dado una posible lista de tareas podría ser:

- Obtener los datos de entrada:
  - Focos
  - Distancia
  - Cantidad de puntos
- Generar punto aleatorio
- Calcular la distancia del punto a los focos de la elipse
- Verificar que el punto generado cumpla con la condición dada
- Añadir los puntos a una lista
- Mostrar la lista por pantalla al usuario

Esta primera aproximación parece cumplir con lo que se solicita en el problema, no obstante, siempre es bueno dar una segunda mirada cuestionando cada parte del proceso.

Por ejemplo, el orden planteado podría no ser lo mejor, o algunas tareas podrían separarse en tareas más pequeñas, o tal vez sería útil tener tareas previas para organizar mejor el trabajo. En este caso en particular un cuestionamiento válido tiene que ver con la generación de puntos aleatorios: ¿dónde busco los puntos? ¿cualquier punto puede servir? ¿habrá una forma más eficiente de determinar los puntos a probar que solo hacerlo aleatorio?

Mirando una elipse, y sabiendo algunas de sus características, es posible encontrar un área de búsqueda más acotada que todo el plano cartesiano o que un área definida arbitrariamente. Particularmente podría ser un cuadrado que encierre a la elipse en su interior, tal como se ve en la imagen (No tiene sentido buscar puntos que pertenescan a la elipse fuera del rectángulo azulo, por ejemplo).



Con eso en mente, y sin tener claro aún como determinar esa área de búsqueda, podemos agregar un par de tareas a la lista:

- Determinar el centro del área de búsqueda
- Determinar el área de búsqueda

La abstracción de procesos corresponde, luego, a la explicación en lenguaje natural (en nuestro caso el idioma español) de qué hará nuestro programa para solucionar el problema. Según lo que hemos hecho hasta ahora nuestra abstracción sería algo como sigue:

Ejemplo

Para solucionar el problema se requiere que el usuario entregue los siguientes datos: los focos de la elipse, dados como pares ordenados, la distancia que determina la constante de la elipse, dada como un entero, y la cantidad de puntos que se espera obtener, también como un entero.

Con esa información se determinará un área de búsqueda en la que podría haber puntos que satisfagan la condición de la elipse, encontrando primero el centro de esta área y luego definiendo sus bordes.

Con el área de búsqueda definido se procederá a generar puntos aleatorios dentro de ella. Por cada punto generado se verificará que la suma de las distancias de dicho punto a los focos de la elipse sea constante, e igual al valor dado por el usuario.

Cada punto verificado se agregará a un listado que, una vez se complete con la cantidad de puntos solicitado, se mostrará al usuario.

Este párrafo describe brevemente como esperamos que nuestro programa solucione el problema solicitado, sin recurrir al lenguaje ni a la sintaxis particular de Python, sino como se le explicaría a alguien que no conoce de programación el paso a paso de la solución[1].

3. Diagrama de abstracción

Para complementar la descripción anterior utilizaremos una versión esquemática de la abstracción de procesos, a la que llamaremos diagrama de abstracción.

El diagrama de abstracción corresponde a una forma gráfica de representar una idea de solución antes de implementarla, tal como la abstracción de procesos es una *forma narrativa de hacerlo*.

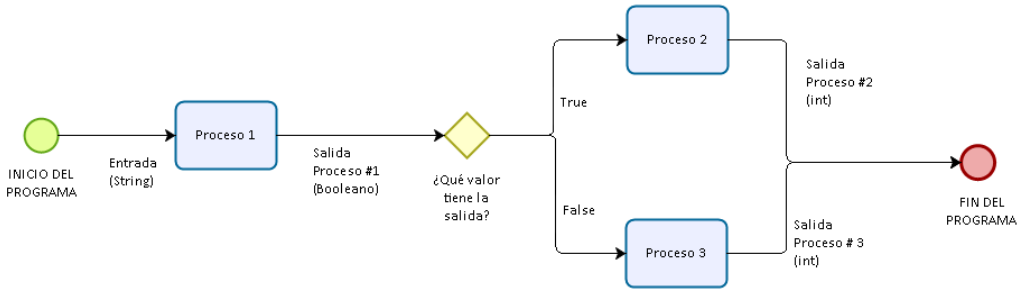
[1] Es importante notar que la abstracción no es infalible, y podría pasar que al implementar la solución descubras que había una forma más clara, sencilla o práctica de resolver parte del problema o su totalidad, por lo que la abstracción debería reformularse para dar cuenta de dicho proceso. A esta revisión y reformualción le decimos que es un proceso iterativo, y que en cada iteración el resultado es mejor que en la anterior, tanto en coherencia como en calidad de la propuesta.

A pesar de que ingeniería existen muchos diagramas para representar distintas abstracciones, en este curso utilizaremos un formato simplificado de un diagrama de procesos para representar abstracciones. Para ello tomaremos la idea de los diagramas de flujo de datos.

En estos diagramas los procesos o subprocesos se representan por un bloque, mientras las entradas y salidas se representan como flechas que entran y salen de los procesos. Cada proceso debe tener a lo menos una salida, y puede tener cuántas entradas se desee o necesite.

Las entradas pueden ser salidas de otros procesos o datos que pueden venir desde afuera (por ejemplo, información que entrega el usuario).

La notación de un diagrama este estilo sería como sigue:



Dónde tenemos tres procesos, que generan cada uno una única salida, y dónde la salida del proceso 1, condiciona la ejecución del proceso 2 o del proceso 3, los cuáles generan la salida del programa.

Es importante notar:

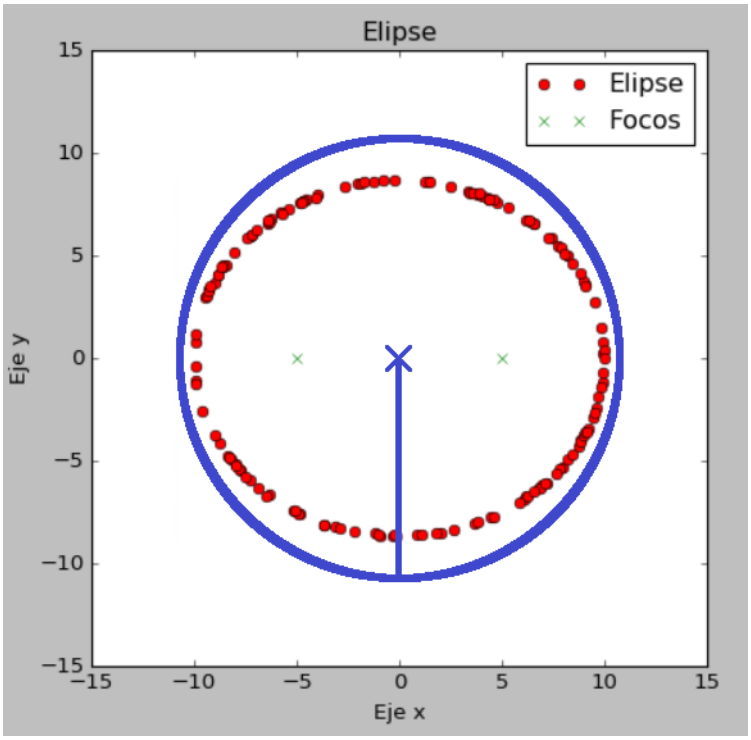
- El diagrama da comienzo al programa con un círculo verde.
- Las entradas y salidas se simbolizan con flechas. Además, debe explicitarse el tipo de dato de la respectiva entrada o salida.
- Los procesos se representan con rectángulos, de borde redondeado, celestes. A efectos prácticos, esperamos que cada uno de ellos represente a una función que hace un trabajo específico. Eventualmente, cada proceso podría tener su propio diagrama si fuese muy complejo.
- Las decisiones, en caso de ser necesario, se representan con un rombo amarillo. Para esta situación las flechas que salen debe relacionarse a las condiciones asociadas: True o False; Mayor, Igual o Menor, etc.
- El diagrama da fin al programa con un círculo rojo.

Ejemplo

Para nuestro ejemplo hay algunas consideraciones que tener en cuenta:

- Considerando las propiedades de las elipses es posible notar que los puntos se distribuyen a lo más dentro de una circunferencia de radio D/2, con D equivalente a la distancia constante de la elipse.

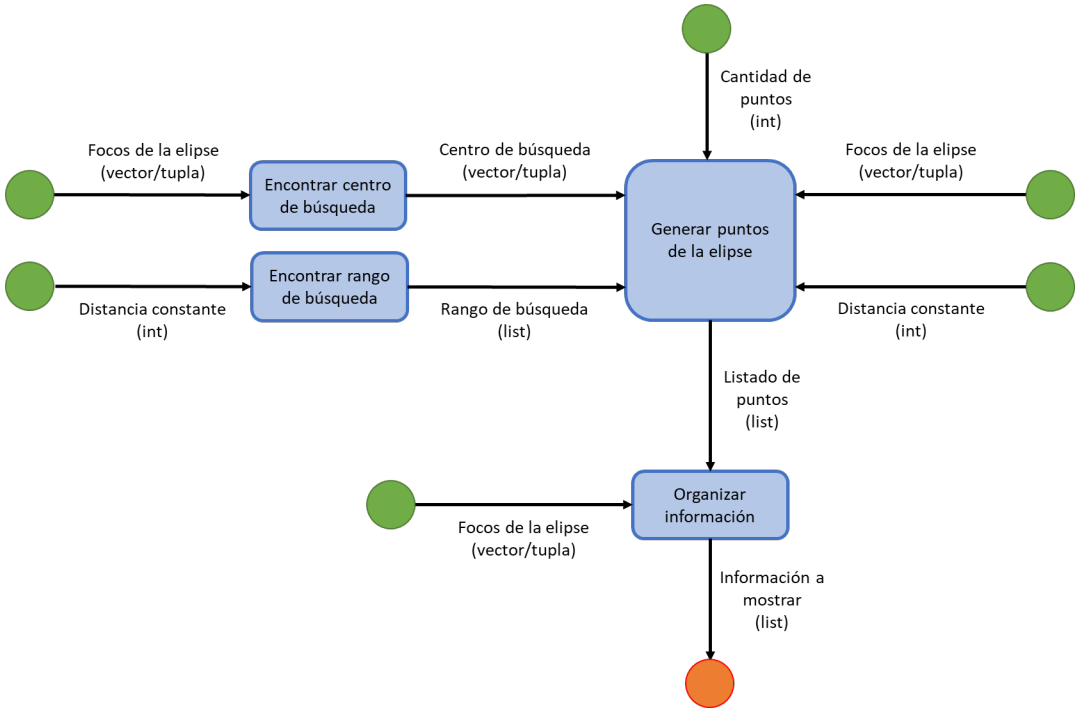
- El centro de dicha circunferencia corresponde al punto medio del segmento que une los dos focos de la elipse.



- La abstracción original consideraba la salida como la lista de puntos directamente, no obstante, esto puede ser poco cómodo de interpretar para un usuario. Dado que el problema no explicita como debe ser la salida, incorporaremos un proceso que muestre más claramente dicha información[2].

[2]Esto corresponde a un ejemplo de lo que sería el proceso iterativo de la abstracción.

Con eso en mente podemos plantear el siguiente diagrama de abstracción:



Es importante notar algunos factores clave a la hora de elaborar un diagrama de abstracción:

- El nivel de detalle: El diagrama debe ser lo suficientemente detallado como para dar cuenta de una solución clara del problema, explicitando el paso a paso en su resolución.
- Los flujos condicionales no siempre se representan: Por ejemplo, al generar los puntos de forma aleatoria muchos de ellos no cumplirán la condición necesaria para ser parte de la elipse y será descartados, no obstante, todo ese proceso queda resumido en "Generar puntos de la elipse". Si las condiciones obligaran a optar entre procesos sería necesario mostrarlas en el diagrama.
- No importa el actor que realiza el proceso: En ningún momento se explicita quién o qué hace las tareas o procesos, sino que se describen en forma general.
- La salida de cada proceso debe ser única: Cada proceso tiene una única salida, no obstante, pueden tener más de una entrada.
- No todas las entradas externas interactúan con el primer proceso: Algunos datos solicitados al comienzo del proceso, como datos de entrada, pueden utilizarse en parte más avanzadas de las tareas, tal como es el caso de la "cantidad de puntos".
- Algunas entradas pueden servir para más de un proceso: Por ejemplo, los focos de la elipse sirven a tres procesos distintos, la distancia constante a dos de ellos, y la cantidad de puntos solo a uno.

4. Implementación de abstracciones

Para implementar la abstracción de procesos diseñada anteriormente, particularmente en su forma de diagrama de abstracción, haremos una equivalencia directa: Por cada proceso dentro del diagrama de abstracción implementaremos una función que hará dicha tarea.

Esta relación nos permitirá organizar nuestro programa siguiendo el diseño del diagrama[3].

A continuación, se muestra la implementación asociada al problema trabajado aquí, explicada paso a paso a través de sus comentarios.

Ejemplo

```
#!/usr/bin/env python
#-*- coding: cp1252 -*-

##################################
# DESCRIPCIÓN DEL PROGRAMA:
##################################

# Programa que encuentra a fuerza bruta una determinada cantidad
    de puntos, solicitados por el usuario, que cumplen con la
    condición de pertenecer a una elipse dada, es decir, que la
    suma de las distancias a dos puntos dados fijos es constante,
    con una cierta tolerancia.


##################################
# IMPORTACIÓN DE MÓDULOS
##################################


import math
import random as rn


##################################
# DEFINICIÓN DE FUNCIONES
##################################


# Función que calcula la distancia entre dos puntos dados, A y B,
     en un plano cartesiano bidimensional.
# Entradas: Dos tuplas de dos elementos cada una correspondientes
     a las coordenadas cartesianas de A y B de la forma (Ax,Ay) y
    (Bx,By), con Ai y Bi números.
# Salida:   Un entero equivalente a la distancia escalar entre A
    y B
def calcularDistanciaEntrePuntos(A, B):

    # Se calcula la distancia entre ambos puntos usando Pitágoras
```

[3] Es importante aclarar que esto no involucra que no puedan agregarse funciones adicionales para facilitar el proceso.

```
    distancia = math.sqrt(abs(A[0] - B[0])**2 + abs(A[1] - B[1])
        **2)

    return distancia


# Función que encuentra un único centro a partir del cual
    realizar la búsqueda de puntos.
# Entradas: Dos tuplas de dos elementos cada una correspondientes
     a las coordenadas cartesianas de cada foco.
# Salida:   Una tupla de dos elementos, equivalente al centro de
    búsqueda.
def encontrarCentroBusqueda(foco1, foco2):

    # Se calcula el punto medio entre los dos focos
    centroBusqueda = ((abs(foco1[0] - foco2[0])/2.0) + min(foco1[
        0], foco2[0]), (abs(foco1[1] - foco2[1])/2.0) + min(foco1[
        1], foco2[1]))

    return centroBusqueda


# Función que genera un rango acotado en donde buscar puntos que
    cumplan con la condición dada.
# El rango servirá para determinar un área cuadrada que
    posteriormente se centrará en el centro encontrado con la
    función anterior.
# Entradas: - Un entero equivalente a la suma de las distancias a
     un punto desde cada foco.
#           - Un entero equivalente a que a la cantidad de veces
    que se desea dividir la unidad.
# Salida:   Una lista de (n*m)+1 elementos, donde n es la
    distancia y m la cantidad de veces que se desea dividir la
    unidad.
def encontrarRangoBusqueda(distancia, variacion=100):

    # Se calcula el rango amplificando la mitad de la distancia
        por la variación tanto a la izquierda del cero como a la
        derecha
    rangoBusqueda = range(-((distancia + 1)/2 * variacion), (
        distancia +1)/2 * variacion + 1)

    return rangoBusqueda


# Función que selecciona puntos aleatoriamente que cumplan con la
     condición de pertenecer a una elipse dada, dentro de un rango
     específico y con un error admisible específico.
# Entradas: - Un entero equivalente a la cantidad de puntos
    deseados.
#           - Dos tuplas de dos elementos cada una
    correspondientes a las coordenadas cartesianas de ambos focos
    de la elipse de la forma (Ax,Ay) y (Bx,By), con Ai y Bi nú
    meros.
#           - Un entero equivalente a la suma de las distancias a
     un punto desde cada foco.
```

```
#           - Una tupla de dos elementos que representa el centro
     del área de búsqueda.
#           - Una lista que representa el rango del área de bú
    squeda.
#           - Un entero que permite definir qué tan fina será la
    posibilidad de variación entre los puntos escogidos; entre
    mayor sea el entero más cerca podrán estar los puntos
    escogidos entre sí.
#           - Un entero que permite definir qué tan cerca de la
    elipse deben estar los puntos escogidos.
# Salida:   Una lista de tuplas, las que equivalen a puntos dados
     en sus coordenadas (Px, Py) que pertenecen a la elipse dada,
    además de incluir los dos focos de esta.
def generarPuntosElipse(cantidad, foco1, foco2, distancia,
    centroBusqueda, rangoBusqueda, variacion=100, tolerancia=0.1):

    # Se calcula la distancia entre los focos
    distanciaEntreCentros = calcularDistanciaEntrePuntos(foco1,
        foco2)

    # Si la distancia entre los focos es mayor la distancia dada
        para la elipse el proceso no puede completarse y se
        informa
    if distanciaEntreCentros >= distancia:
        return (False,distanciaEntreCentros)

    # Si no, se comienza la búsqueda de los puntos respectivos
    else:

        # Se define una bandera o indicador que permite comenzar
            un ciclo
        verificador = True

        # Se define una lista vacía donde se almacenarán los
            puntos encontrados
        puntos = []

        # Se da inicio al ciclo de búsqueda
        while verificador:

            # Para cada coordenada, con centro en el centro único
                 definido antes y con el radio de búsqueda
                apropiado calculado antes se procede, mediante
                fuerza bruta, a generar los puntos aleatorios
            puntoX = centroBusqueda[0] + rn.choice(rangoBusqueda)
                /float(variacion)
            puntoY = centroBusqueda[1] + rn.choice(rangoBusqueda)
                /float(variacion)
            punto = (puntoX, puntoY)

            # Se verifica si el punto no está en la lista, para
                evitar repeticiones y si la diferencia entre la
                distancia dada y la suma de las distancias del
                punto generado a los focos es menor a la
                tolerancia dada
            if  (not (punto in puntos)) and abs(
```

```
                calcularDistanciaEntrePuntos(punto,foco1) +
                calcularDistanciaEntrePuntos(punto,foco2) -
                distancia) <= tolerancia:

                # Si se cumple la condición el punto es válido y
                    se agrega a la lista de puntos
                puntos.append((puntoX,puntoY))

            # Si no es válido se omite
            else:
                pass

            # Se verifica la cantidad de puntos añadidos
            # Cuando la cantidad alcanza la solicitada se cambia
                la bandera o indicador a False, para romper el
                ciclo, sino se continúa
            if len(puntos) == cantidad:
                verificador = False
            else:
                pass

        # Se agregan los puntos que representan a los focos de la
             elipse
        puntos.append(foco1)
        puntos.append(foco2)

        return puntos


# Función que organiza la información a mostrar en una lista que
    incluye, en cada elemento, el punto encontrado, la distancia
    al foco 1, la distancia al foco 2 y la suma de ambas
    distancias.
# Entradas: - Lista con los puntos encontrados.
#           - Dos tuplas de dos elementos cada una
    correspondientes a las coordenadas cartesianas de cada foco.
# Salida:   Una tupla de dos elementos, equivalente al centro de
    búsqueda.
def organizarInformacion(listadoPuntos, f1, f2):
    listaOrdenada = []
    for punto in puntosGenerados:
        # Para cada punto en la lista calculamos su distancia a
            cada foco
        distanciaF1 = calcularDistanciaEntrePuntos(punto,f1)
        distanciaF2 = calcularDistanciaEntrePuntos(punto,f2)
        # Organizamos la información a mostrar
        datos = str(punto)+" - Dist a F1: "+str(distanciaF1)+" -
            Dist a F2: "+str(distanciaF2)+" - Dist Total: "+str(
            distanciaF1 + distanciaF2)
        listaOrdenada.append(datos)
    return listaOrdenada



################################
# Bloque Principal
```

```
################################


################################
# Entrada

# Se solicita al usuario todos los datos necesarios
focoA = raw_input("Ingrese, separados por una coma, los valores
    de las coordenadas del primer centro (Ej. 0,0): ").split(",")
focoB = raw_input("Ingrese, separados por una coma, los valores
    de las coordenadas del segundo centro (Ej. 10,10): ").split(",
    ")
distanciaCostante = raw_input("Ingrese la distancia constante
    equivalente a la suma de las distancias a cada centro: ")
cantidadDePuntos = raw_input("Ingrese la cantidad de puntos que
    desa obtener: ")


################################
# Procesamiento

# Se transforma cada datos de entrada en flotantes o enteros segú
    n corresponda
focoA = (float(focoA[0]), float(focoA[1]))
focoB = (float(focoB[0]), float(focoB[1]))
distanciaCostante = int(distanciaCostante)
cantidadDePuntos = int(cantidadDePuntos)

# Ubicamos el centro de búsqueda
centro = encontrarCentroBusqueda(focoA, focoB)

# Estimamos el rango de búsqueda
rango = encontrarRangoBusqueda(distanciaCostante)

# Encontramos los puntos que cumplen la condición dada
puntosGenerados = generarPuntosElipse(cantidadDePuntos, focoA,
    focoB, distanciaCostante, centro, rango)

# Organizamos la información para mostrarla
datosMostrar = organizarInformacion(puntosGenerados, focoA, focoB
    )

################################
# Salida

# Con el uso de un ciclo recorremos la lista de datos organizada
    y se imprime a pantalla elemento a elemento
i = 0
while i < len(datosMostrar)-2:
    print datosMostrar[i]
    i += 1
```

Actividad

Responde las siguientes preguntas:

- ¿Qué hace el =100 en def encontrarRangoBusqueda(distancia, variacion=100)?
- ¿Por qué es necesario incluir la variación en la generación de rangos?
- ¿En qué afecta variar el valor de la tolerancia en el programa?

Ejercicio Propuesto

Realice el mismo procedimiento que hicimos para la elipse, ahora para *"el lugar geométrico de todos los puntos que equidistan de un único punto"*.