La recursividad es una forma declarativa para definir procesos que se repiten, tienen definición formal en matemáticas (Explicado mas adelante), en este sentido es similar a los ciclos de control imperativo como for y while.

A diferencia de las secuencias de control tradicionales, la recursividad aplica a nivel de funciones, se habla entonces de funciones recursivas (o no recursivas que son las que se han trabajado hasta ahora), cualquier cosa que se pueda hacer con for o while puede lograrse con funciones recursivas

Pero no todo lo que se puede hacer con recursividad se puede hacer con for y while (o resulta ser demasiado complejo), lo que lo vuelve una forma de repetición mas general y poderosa. En términos simples una función recursiva es aquella que se define en términos de ella misma, esto es en algún punto de su propio cuerpo se manda a llamar a sí misma (también puede ser de forma indirecta como se mencionara más adelante)

Def a():  
 a()

A muchos programadores se les dificulta el uso de recursividad, incluso a programadores profesionales, esto se debe a su naturaleza declarativa, siendo que la programación se enseña normalmente de forma imperativa. En términos generales puede verse a la recursividad como una técnica de programación que permite resolver problemas que de otra forma resultan muy difíciles o imposibles de resolver con las estructuras de control tradicionales (se verán ejemplos concretos de esto en el tema de árboles)

**Programación imperativa:**

Se refiere a darle a la computadora paso por paso las instrucciones que debe ejecutar, es trabajo del programador dar todos los detalles del control del programa, esto es, el programador debe definir como resolver el problema (el tipo de programación que he usado

**Programación declarativa:**

El programador se preocupa principalmente por definir el problema, esto es, el programador define que es el problema. Existe un mecanismo de resolución automático que recibe definiciones de problemas y genera soluciones, este mecanismo puede ser por ejemplo un motor de resolución lógica (usado en programación lógica), todos los sistemas operativos tienen un motor implícito de resolución de problemas recursivos, este funciona a través de la pila de memoria del proceso (se explicara mas adelante)

**Ejemplo imperativo**

Def Fibonacci(numero):  
 if numero <3:  
 return 1  
anterior = 1  
actual = 1  
for \_ in range(numero-2):  
 aux = actual  
 actual = anterior + actual  
 anterior = aux  
return actual  
  
print(Fibonacci(10)) #imprime 55

**Ejemplo declarativo**

Def Fibonacci(numero):  
 if numero < 3:  
 return 1  
return Fibonacci(numero-2) + Fibonacci(numero-1) #está sumando el número anterior y ante anterior del número actual, en este caso, 10  
  
print Fibonacci(10)) #imprime 55

Estas suelen ser más cortas y elegantes que las imperativas, pero suelen resultar más abstractas y difíciles de entender (sobre todo si no estas acostumbrado), en un curso posterior (lenguajes y paradigmas de programación) se hablara mas del tema de programación declarativa

**Manejo de memoria**

La memoria RAM es una memoria de acceso rápido (a diferencia del disco duro), básicamente se utiliza para tener acceso a los datos de los programas que se encuentran en ejecución (llamados procesos), todo programa en ejecución esta cargado en la RAM (al menos parcialmente si no hay suficiente memoria); la memoria RAM puede verse como una matriz de celdas, cada celda guarda un byte y cada celda tiene una direccion de memoria definida lo que permite que el acceso sea aleatorio, no es necesario recorrer otras celdas para encontrar la celda de interés.

**Espacios de memoria en un proceso**

Al ejecutarse un proceso, el sistema operativo reserva espacio en memoria para guardar los datos del proceso, tienen 4 áreas de memoria principales (aunque puede variar entre sistema operativos), estos son: **CALL STACK, HEAP, DATOS Y EL CODIGO**

**CALL STACK**

También llamado simplemente stack, es una zona de memoria que se comporta como una pila, su función principal es   
También en varios lenguajes (no en caso de Python) es en esta zona donde se guardan por defecto las estructuras de datos creadas dentro de una función, en esta memoria también se propagan los valores retorno entre funciones, si un proceso excede su tamaño de pila asignado, se produce un error de desbordamiento de pila o stack overflow, esto puede suceder si se anidan demasiadas llamas de funciones (varios de miles) un desbordamiento de pila provoca la interrupción del programa, el programador no puede afectar directamente al call stack, los datos de esta zona se crean y destruyen automáticamente por el sistema operativo, y esta zona de memoria es la que permite que haya recursividad

**Heap**

Es un área especial del proceso designada para el uso de los programadores, en algunos lenguajes como C el programador puede reservar y borrar memoria en esta área, el sistema operativo no limpia esta memoria (como si lo hace con el stack), en lenguajes de mas alto nivel como Python, el programador no toma decisiones sobre donde colocar datos, se utiliza la opción mas robusta de forma transparente, por ejemplo, en Python al crear una lista dentro de una función se usa automáticamente el heap, de esta forma la función regresa una referencia a la lista, no se corre el riesgo de que la memoria asociada a la lista sea limpiada de forma automática al terminar la función (como si pasaría si se guardara en el stack).  
Como se menciono en temas anteriores, en lenguajes como Python se utiliza un recolector de basura para limpiar memoria que no se utiliza, mas concretamente la limpieza se hace sobre el heap (dado que de por si el stack se limpia solo)

**Datos**

Es un espacio especial para almacenar variables globales y constantes, es un espacio fijo que se designa en tiempo de compilación (en lenguajes compilados)

**Codigo**

En este espacio se almacena el código del programa asociado al proceso, de esta forma se puede llevar un control para que el procesador sepa que instrucción debe ejecutar a continuación, es importante recordar en que punto se encuentra la ejecución dado que los procesos entran y salen constantemente del procesador (el calendarizador del SO se encarga de esta función)

**Funciones recursivas**

Como ya se mencionó, una función recursiva es aquella que se define en términos de ella misma, un ejemplo común es la función factorial.

Las funciones recursivas tienen casos base y casos recursivos, un caso base es una entrada de la función para la cual se puede entregar directamente un valor, un caso recursivo es aquel que hace referencia a la propia función (hace llamada a la propia función).

def factorial(numero):  
 if numero == 0:  
 return 1  
return numero \* factorial(numero-1)  
print(factorial(6)) #imprime 720

**Como funcionan**

Cada vez que se hace una llamada recursiva se apila llamada en el stack, quedando pendiente la finalización de la función, al alcanzar un caso base se empieza a desapilar propagando resultados en la pila, dado que se trabajo con una pila (como se vio en el tema de pilas) las cosas suceden en el orden inverso después de la llamada recursiva, por ejemplo, una manera sencilla de imprimir del 5 al 1:

def imprimir\_inverso(n, máximo):

if n <= máximo: # caso base implícito

imprimir\_inverso(n + 1, máximo) # primero la llamada recursiva

print(n)

imprimir\_inverso(1, 5)

**Diseño de soluciones recursivas**

Un error que cometen muchos programadores al crear funciones recursivas es pensar constantemente en lo que pasa en el stack, no se debe hacer eso: es importante saber cómo funcionan las llamadas recursivas, pero pensar en ello aumenta la carga cognitiva y la complejidad, En vez de eso debes pensar en los casos, esto es programación declarativa, no hay que pensar en el proceso detallado, sólo en definir el problema, El manejo de la pila de llamadas es tu mecanismo de resolución automático, no es necesario pensar en cómo funciona

**Consejos**

Empieza por los casos base, ¿En qué casos puedo regresar un resultado directamente?  
Luego piensa en los casos recursivos, ¿Qué debo hacer cuando no se puede resolver el caso, cómo debería tratarse?  
Considera que los parámetros que recibe tu función son muy importantes, hay que pensar muy bien en ellos  
En los casos recursivos los parámetros que pasas deberían variar de cierta forma, piensa bien en cuál es esa forma  
Si estás procesando listas, en cada llamada evalúa el elemento del frente y pasa el resto de la lista en los casos recursivos  
Muchas veces es mejor definir dos funciones: una recursiva y otra no recursiva que sólo manda a llamar a la función recursiva.  
La función recursiva puede requerir que se inicialicen parámetros de forma especial, cosa que se hace desde la función no recursiva  
También puede aprovecharse la función no recursiva para copiar memoria si es que la función recursiva modifica memoria

**Tipos de recursividad**

**Directa**

Es la forma que se ha visto hasta el momento, se refiere a que en el cuerpo de la función se mande a llamar la propia función.

**Indirecta**

Cuando dos o mas funciones se hacen llamadas entre si, es menos común y útil que la directa

def a(i):

if i == 0:

return True

return b(i-1)

def b(i):

return a(i)

**Estilos de recursividad**

**A la cola**

En general es el estilo mas declarativo, por lo tanto suele dar soluciones cortas, se basa en dejar trabajo pendiente (en cola de espera, aunque en realidad sabemos que se usa una pila), esto es, se hacen llamadas recursivas y luego se continua con el código, este estilo no suele requerir agregar parámetros extra a la solución, por lo que tampoco es en muchas ocasiones necesario agregar una función no recursiva que inicialice los parámetros de la función recursiva. Es ineficiente en memoria y procesador dado que requiere de apilar más trabajo

def mayor\_lista(lista):

if not lista:

return -1

frente = lista[0]

resto = lista[1:]

if not resto: # la lista tenía un solo elemento

return frente

mayor\_resto = mayor\_lista(resto) # trabajo pendiente adelante

if frente > mayor\_resto:

return frente

else:

return mayor\_resto

**No a la cola**

Es en cierto sentido similar a un for y a un while, lo ultimo que se hace en los casos recursivos es la llamada recursiva, esto es, no deja trabajo pendiente. Normalmente requiere de agregar parámetros extra de control, por ejemplo, índices o acumuladores de resultados. En algunos lenguajes de programación, aunque no es el caso de Python, como Common Lips, este tipo de recursividad no apila llamadas en la pila (se saca de la pila la función que esta terminando, por lo que no puede caer en un stack overflow) En esos lenguajes la recursividad no a la cola es muy similar en terminos de eficiencia a un for o while tradicional, en general es mas eficiente que la recursividad a la cola

def mayor\_rec(lista: list, mayor: int) -> int:

"""

Regresa el número mayor, requiere inicialización.

returns: int

"""

if not lista: # lista vacía

return mayor # mayor es un acumulador

frente = lista.pop()

if frente > mayor:

return mayor\_rec(lista, frente) # es lo último que se hace

return mayor\_rec(lista, mayor) # es lo último que se hace

# se requiere de una segunda función de inicialización

def mayor\_lista(lista: list) -> int:

"""

Regresa el mayor de la lista.

lista: list

returns: int

"""

lista = lista[:]

if not lista:

return -1

mayor = lista.pop()

return mayor\_rec(lista, mayor)