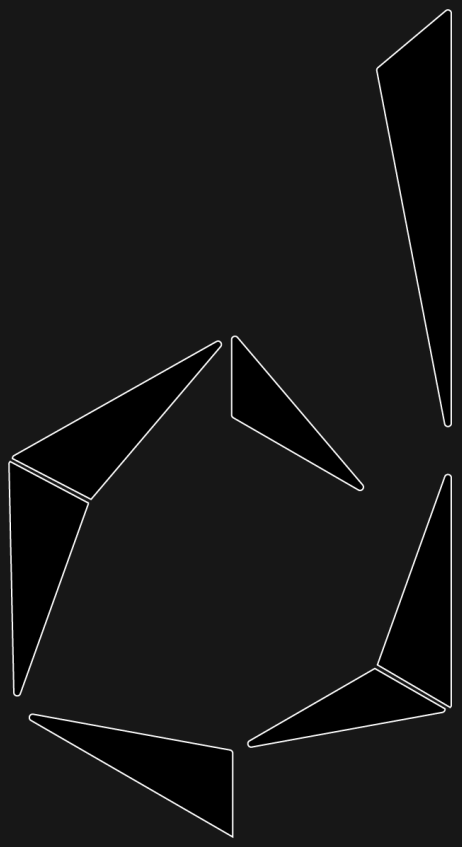
INGENIERÍA MECATRÓNICA



DI_CERO

DIEGO CERVANTES RODRÍGUEZ

PROGRAMACIÓN: MANEJO DE DATOS Y RECURSOS

PYTHON 3.9.7, DJANGO, DOCKER, KUBERNATES, LINUEX, ETC.

Estructuras de Datos: Repaso Conceptos

Contenido

Estructuras de Datos

- **TIPOS DE ESTRUCTURAS DE DATOS EN PYTHON:** La gran diferencia entre su clasificación, es que algunos de ellos tienen cierto orden (índice y valor) y otros no, además de que algunos son editables o mutables, donde se les puede agregar, eliminar, o modificar elementos y otros son inmutables, donde sus datos no se pueden cambiar una vez establecidos.
    - **Listas (list[]):** Una lista es una colección **ordenada, heterogénea (varios tipos de datos) y mutable (editable)** de elementos. Se definen utilizando corchetes [] y podemos convertir una **tupla, diccionario o conjunto** a una lista con el método **list()**.

```
Ejemplo: lista = [1,2,"UPIITA",True]
lista[0]; #1. lista[2]; #"UPIITA". lista[-2]; #"UPIITA".
lista[-1] #True.
lista[-1] = [3,4,False] #lista = [1,2,"UPIITA",[3,4,False]].
```

    - **Tuplas (tuple()):** Una tupla es una colección **ordenada, heterogénea e inmutable** de elementos. Se definen utilizando paréntesis () y podemos convertir una **lista, diccionario o conjunto** a una tupla con el método **tuple()**.

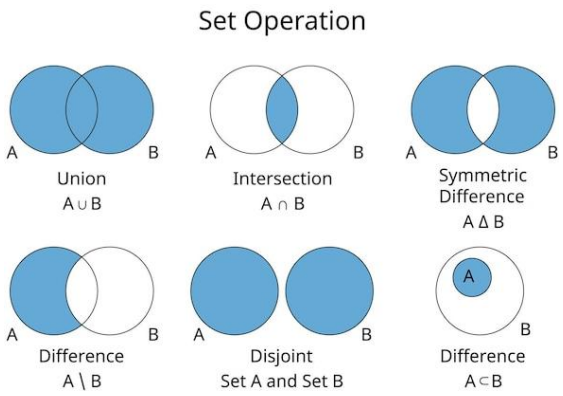
```
Ejemplo: tupla = (1,2, "UPIITA",True); tupla2 = ("olis",);
tupla[0]; #1. tupla[-1]; #True. tupla2[0]; #"olis".
```

    - **Diccionarios (dict{}):** Un diccionario es una **colección desordenada, heterogénea y mutable** de pares **clave-valor**. Se definen utilizando llaves {} y separando cada par clave-valor por dos puntos (:), de la misma forma como se hace cuando se crea un JSON. Cabe mencionar que **el valor de un diccionario no a fuerza debe ser un número o un string, puede ser una lista, tupla o conjunto**, pero su clave debe ser única y su valor debe ser forzosamente nativo o inmutable **(int, float, str, iterador zip(), tuplas, etc.).** También estos pueden ser convertidos a un diccionario con el método **dict()**.

```
Ejemplo: diccionario = {'Café':["Cafeto","Bebida","Color"]};
diccionario['Café']  #["Cafeto","Bebida","Color"].
dict2 = {1:"Uno",2:"Dos"}; dict2[1]; #"Uno".
dict2[3] = "Tres";   #dict2 = {1:"Uno",2:"Dos",3:"Tres"}.
dict2[3] = "Cuatro"; #dict2 = {1:"Uno",2:"Dos",3:"Cuatro"}.
```

    - **Conjuntos (set{}):** Un conjunto es una **colección desordenada, heterogénea y mutable** de elementos únicos. **No permite elementos duplicados** y no tiene un orden definido, tampoco puede contener **listas, diccionarios, ni conjuntos** anidados **mutables**, solamente valores nativos o inmutables **(int, float, str, iterador zip(), tuplas, etc.).** Se definen con el método **set()** y así como los **diccionarios**, utilizan las llaves {}.
        - Estas estructuras de datos están relacionadas con un concepto muy importante que tocaremos después llamado **hash**, relacionado a la **complejidad algorítmica**.
            - Con los conjuntos se pueden realizar operaciones lógicas de intersección (**conjunto_1.intersection(conjunto_2) o conjunto_1 & conjunto_2**), no inclusión (**conjunto_1.issubset(conjunto_2)**), etc.

```
Ejemplo: conjunto = set((1,"hola",(1,2),1)) #{1,"hola",(1,2)}.
conjunto = set([1,"hola",(1,2),1]) #{1,"hola",(1,2)}.
conjunto = set("hola") #{'h','o','l','a'}.
```



- **ESTRUCTURAS ANIDADAS:** Cuando dentro de la posición de una **lista**, **tupla**, **diccionario** o **conjunto** se encuentre otra estructura interna, se le llama **estructura anidada**, esto se realiza por ejemplo para categorizar datos, realizar operaciones matriciales con diferentes dimensiones, etc. Las estructuras anidadas pueden ser del mismo tamaño o de distinto tamaño y para introducir o editar un dato, debemos indicar los índices de la dimensión a donde queremos acceder a través de la sintaxis **nombreEstructura[index_dimension1]…[index_dimension_n]**:

```
▪ Ejemplo: lista_anidada = [[1,2,3,4],[1,5,6]]
▪ Ejemplo: lista_anidada_2 = [[1, [8,9,10], 3], [4,5,6]]
  lista_anidada_2[1][2] = [10,"UPIITA",11,12]
  #lista_anidada_2 = [[1, [8,9,10], 3], [4,5, [10,"UPIITA",11,12]]]
```

- **SUBESTRUCTURAS (SLICING):** Podemos crear una subestructura a través de indicar rangos de índices y guardándolos en una variable nueva, primero colocando el inicial y luego el final (denotando que el último índice no se alcanzará), a través de la siguiente sintaxis **nombreEstructura[index_inicial : index_final_no_incluido]**, y recordando que los índices positivos de las estructuras de datos se cuentan desde 0 hasta el final de sus elementos "n-1", además de que podemos recurrir a los índices negativos, donde se cuenta desde -1 (siendo el último elemento), hasta "-n", siendo el primero. También a través del **SLICING** se puede extraer y modificar a la vez los elementos de una lista.
  - **SLICING CON PASO:** También existe el slicing (extracción de datos), pero con paso, indicando así, el índice inicial y final no alcanzado, pero de igual forma al final se indica cuantos elementos se brincará para extraer el siguiente dato, a esto se le llama paso y la sintaxis es la siguiente: **nombreEstructura[inicial : final_no_incluido: paso]**.

```
▪ Ejemplo extracción: lista = [1,2,"UPIITA",True]
▪ sublista = lista[2:4]     #sublista = ["UPIITA",True];
  sublista = lista[:-1]     #sublista = [1,2, "UPIITA"];
  sublista = lista[::-1]    #sublista = [True,"UPIITA",2,1];
▪ Extracción y modificación (slicing): lista[2:4] = ["olis",False];
▪ #lista = [1,2, "olis",False]
```

Nota: Si queremos acceder a un índice positivo o negativo donde no existe ningún dato, porque nuestro tamaño de estructura de datos es menor, obtendremos un error tipo: ***index out of range****.*

Los métodos más utilizados con los distintos tipos de estructuras de datos en Python son los siguientes y muchos de estos utilizan algo llamado **índice**, *esto se refiere a una coordenada que le podemos pasar a*

la estructura para acceder a una de sus posiciones y así editar, agregar o eliminar un dato. Lo importante a considerar del **índice** y del **tamaño** de las estructuras de datos, es que, si una **lista** tiene **10 datos**, **su longitud es de 10**, pero accederemos a sus **posiciones** a través de los números 0 a 9:

- **len(lista, tupla, diccionario o conjunto):** El método **len** nos devuelve un valor entero que representa el **tamaño** de una estructura de datos, donde si una **lista** tiene **10 datos**, **su longitud es de 10, pero tendrá 9 índices, de 0 a n-1**. Si la estructura tiene estructuras anidadas, también podemos calcular su **tamaño**, indicando su índice (**lista** o **tupla**), clave (**diccionario**) o el número total de sus elementos (**conjunto**).
- **lista.append(), diccionario.update({key: value}) o conjunto.add()**: Los métodos **append()**, **update()** o **add()** sirven para introducir un dato nuevo a una estructura de datos **mutable** hasta el final de sus posiciones.
- **lista.count(dato), tupla.count(dato), list(diccionario.keys_o_values()).count(dato)**: El método **count(dato)** sirve para contar el número de veces que se repite un elemento en una **lista, tupla o diccionario**, ya que en los **conjuntos** no existen elementos repetidos. Su ejecución es de tipo lineal **O(n)** (veremos que significa esto después en el documento) y no hay forma directa de ejecutarlo en **diccionarios**, sino que antes estos deben ser convertidos a **listas o tuplas**, para después contar las veces que aparece una **clave o valor** específico, dependiendo si utilizamos el método **keys()** o **values()** durante la conversión.
- **lista.index(dato) o tupla.index(dato)**: El método **index(dato)** nos devuelve el índice de la primera aparición de un dato en una estructura ordenada (solo **listas o tuplas**, no **diccionarios o conjuntos**) y un error de tipo **ValueError** si el dato no existe.
    - **diccionarios[key]**: En las estructuras no ordenadas, osea aquellas que no tienen ningún índice, se accede a sus datos a través de una estructura de clave-valor **(key-value)**, con una ejecución con complejidad algorítmica constante **O(1)** por medio de un elemento llamado **hash**, el cual nos permite acceder directamente a los valores asociados a una key específica directamente, sin recorrer los demás elementos.
- **lista.remove(valor), conjunto.remove(valor), lista.pop(índice), diccionario.pop(key), o conjunto.discard(valor)**: El método **remove()** sirve para eliminar la primera aparición de un valor, devolviendo None si elimina correctamente el elemento, pero retornando un error de tipo **ValueError o KeyError** si el dato no existe; a su vez, **discard()** hace lo mismo pero con la diferencia de que no retorna **Ningún error** si el dato no existe, mientras que el método **pop()** recibe el índice **(lista)** o key **(diccionario)** y devuelve el valor del dato eliminado.
- **sorted(lista, diccionario.keys_o_values() o conjunto):** El método **sorted** sirve para ordenar de menor a mayor los elementos numéricos de una **estructura mutable (editable),** como una **lista, diccionario, conjunto** etc.
- **zip(lista, tupla):** Método que une varias estructuras de datos iterables posición por posición, creando un tipo de dato llamado iterable, el cual son **mini tuplas** que unen los valores de las estructuras unidas en una sola.
    - Los iterables no funcionan por sí solos, por eso debemos convertirlos a una **lista, tupla, diccionario o conjunto** posteriormente.
        - Ejemplo:
```
letras = ['g', 'o', 'l']
freq   = [2, 3, 1]
iterable = zip(letras, freq)
```

- **list(), tuple() o dict():** Como los iterables no pueden ser manejados por sí solos, forzosamente deben ser introducidos a una nueva estructura de datos, como una **lista**, **tupla**, **diccionario** o **conjunto**. En el caso de las listas o tuplas, el iterador se vería como una lista anidada y en los diccionarios o conjuntos sus datos se ordenarían con el formato de clave-valor, donde su primer parámetro indicará su key y el segundo su value **{"key": value}**.

- **diccionario.items():** Método que devuelve una **lista** de **tuplas** con los **pares de datos {key: value}** dentro de las **tuplas internas** de la **lista** = **[(key, value)]**.

Condicionales y Bucles

Condicionales if y else if (elif)

En Python no se utilizan llaves de apertura o cierre al utilizar condiconales, solamente se utilizan dos puntos para indicar el inicio del condicional y tabuladores para ver qué es lo que está dentro o fuera de él, ya sea para el condicional if, else if (elif) o else.

- **CONDICIONAL IF:** A través de la sintaxis **if(condicion):** y utilizando comparadores (<, >, ==, etc.) u operadores lógicos (and, or, not, etc.) se indica una condición, donde si esta es verdadera, se ejecuta el código que haya dentro del condicional, sino todo esto se brinca y se pasa a la línea siguiente del condicional.

```
if(n == 0 or n == 1):
   #Acción que se ejecuta si la condición es true.
```

- **CONDICIONAL ELIF:** Este condicional funciona de forma muy similar al condicional if, con la diferencia de que se evalúan varios condicionales que estén relacionados entre sí, muy utilizado para evaluar rangos, donde podemos evaluar por ejemplo si un número se encuentra en el rango 0 a 10, 10 a 20 o 20 a 30.

```
elif(0 <= n <= 10):
   #Acción que se ejecuta si la condición es true.
elif(10 <= n <= 20):
   #Acción que se ejecuta si la condición es true.
elif(20 <= n <= 30):
   #Acción que se ejecuta si la condición es true.
else:
   #Acción que se ejecuta si ninguna de las condiciones anteriores se cumple.
```

- **CONDICIONAL ELSE:** Esta sección de los condicionales if o elif indica la acción que se debe ejecutar cuando la condición no se cumpla, cubriendo así todas las posibilidades de ejecución.

Bucler for y while

- **BUCLE FOR:** Este es un bucle finito que se ejecutará un número determinado de veces, dado por la sintaxis **for variable in range(0, n+1)**, donde **i** es una variable local numérica que solo existirá

dentro del bucle, la cual adoptará el rango de valores dados por **range(0, n+1)**, donde se indica el valor de inicio de la ejecución y su valor final, que nunca será alcanzado, ya que el bucle se detendrá antes, osea que se ejecutará **n** veces, no **n+1**.

```
for i in range(2, n+1):    #La fórmula del factorial es: n! = 1*2*3*...*(n-1)*n
   factorial = factorial*i #Se pueden usar para sumatorias matemáticas Σ.
```

- Los bucles se pueden utilizar para leer individualmente los elementos de una estructura de datos, ya sea para pasarlos a otra estructura de datos o para realizar operaciones con ellos, por ejemplo:
  - **Multiplicación de matrices:**

```
#cij = ai*bj = Σai*bj
m = len(A) #Número de filas o renglones de la matriz A
n = len(B[0]) #Número de las columnas de la matriz B
l = len(A[0]) #l = len(A[0]) = len(B)
for i in range(0, m): #Lectura de filas de la matriz A.
    for j in range(0, n): #Lectura de columnas de la matriz B.
        sum = 0 #Inicialización de la variable que almacenará el valor de cada elemento de la matriz resultante.
        for k in range(0, l): #Lectura de las columnas de la matriz A, que es igual a las filas de la matriz B.
            sum = sum +  A[i][k]*B[k][j] #Llenado de los elementos de la matriz C: cij = ai*bj = Σai*bj
        #for k
        C[i][j] = sum #Asignación del valor de cada elemento de la matriz resultante C.
    #for j
#for i
```

  - **Extraer las letras de un string:**

```
st = "google.com"
letras = []
for i in range(len(st)):    #Como un string es una cadena de caracteres, también sirve poner: for i in st:
    letras.append(st[i])    #También sirve utilizar: letras.append(letra)
letras = ['g', 'o', 'o', 'g', 'l', 'e', '.', 'c', 'o', 'm']
```

  - **Crear un diccionario con frecuencia de letras repetidas:**

```
st = "google.com" #Palabra de ejemplo a analizar
stLenght = len(st)
lastChar = ""
#Vector con una longitud igual al número de caracteres de la palabra de ejemplo, donde se guardan los índices en
#donde por primera vez apareció un carácter en la palabra analizada.
chIndex = []
for i in range(0, stLenght):
    #Guarda de uno en uno los caracteres correspondientes a los índices de 0 hasta el tamaño de la palabra de ejemplo.
    lastChar = st[i]
    #En el bucle for anidado se recorre caracter a caracter el string. Si st[i] = st[j], osea que la letra identificada
    #en el primer bucle for sea igual a la reconocida en el segundo bucle for, se rompe el ciclo anidado for con
    #índice j y se guarda el valor numérico del índice j en el vector chIndex, pasando así a analizar el siguiente
    #caracter de st.
    for j in range(0, stLenght):
```

```
        if(lastChar == st[j]): #st[i] = st[j], osea que la letra analizada en el primer y segundo for sean iguales
            chIndex.append(j) #append(): El método append() sirve para agregar valores a una lista, array o diccionario
            break #Se rompe el ciclo anidado for con índice j
chIndex = [0,1,1,0,4,5,6,7,1,9]
chIndexSort = sorted(chIndex)          #sorted(): Ordenar de menor a mayor los índices.
chNewIndex = set(chIndexSort)               #set(): Eliminar los índices que se repiten, de los que ya estaban ordenados.

chIndexSort = [0,0,1,1,1,4,5,6,7,9]
chNewIndex = [0,1,4,5,6,7,9]
freq = [] #Vector donde se guarda la frecuencia de cada letra analizada perteneciente a la palabra original.
for i in range(0, len(chNewIndex)):
    sumFreq = 0
    for j in range(0, len(chIndexSort)):
        #Comparación de los índices de la lista chIndexSort y chNewIndex para saber la frecuencia de cada letra.
        if(chIndexSort[j] == chNewIndex[i]):
            sumFreq += 1 #Se suma un 1 a la variable sumFreq cada que sea igual un índice de ambas listas
    freq.append(sumFreq) #append(): El método append() sirve para agregar valores a una lista, array o diccionario.
freq = [2,3,1,1,1,1,1]

#Palabra de ejemplo a analizar: st = "google.com"
letras = [] #Vector donde se guardarán las letras de los índices no repetidos de la palabra original.
#Bucle for para crear la lista letras que contiene todas las letras no repetidas en la palabra
for i in range(0, len(chNewIndex)):
    letras.append(st[chNewIndex[i]])
letras = ["g","o","l","e",".","c","m"]

#zip(): Método que crea un iterable con dos listas o tuplas del mismo tamaño.
#dict(): Método que convierte un iterador en un diccionario, con formato clave, valor.
letras_y_frecuencias= dict(zip(letras, freq))
letras_y_frecuencias = ["g":2,"o":3,"l":1,"e":1,".":1,"c":1,"m":1]
```

- **Método más sencillo de frecuencia con JSON (diccionarios) directamente:**

```
diccionario = {}
    #Los bucles for-each son muy buenos para recorrer elementos de una lista o diccionario, ya que los strings
    #son cadenas de caracteres, por lo tanto, en vez de poner for i in range(len(st)), podemos poner for i in st.
    for letra in st:    #Bucle for que recorre cada letra del string que reciba.
        #print(): Imprimir un mensaje en consola.
        print(letra)    #Imprime las letras del string una por una.
        #diccionario.keys(): Con este el método keys() que se aplica a un diccionario obtenemos todas sus keys,
        #donde la estructura de los diccionarios se asemeja a los JSON: {"key": "value"}, para acceder a los
        #values de un json, debemos indicar su key, ya que no tenemos indices en este tipo de estructuras.
        llaves = diccionario.keys()
        #Lo que hace aquí el JSON es comparar cada letra del string, con un diccionario que las va recorriendo
        #una a una, cuando encuentre que existe una vez, le sumará un 1 y lo asignará como su value, indicando
        #su frecuencia, entonces:
        # - En la iteración 1 donde: letra = g; diccionario = {};  diccionario[letra] = diccionario[g] = None; diccionario = {'g':1}
        # - En la iteración 2 donde: letra = o; diccionario = {'g':1};  diccionario[o] = None; diccionario = {'g':1,'o':1}
        # - En la iteración 3 donde: letra = o; diccionario = {'g':1,'o':1};  diccionario[o] = !None,+=1; diccionario = {'g':1,'o':2}
        # - En la iteración 4 donde: letra = g; diccionario = {'g':1,'o':2};  diccionario[g] = !None,+=1; diccionario = {'g':2,'o':2}
        # - En la iteración 5 donde: letra = l; diccionario = {'g':2,'o':2};  diccionario[l] = None; diccionario = {'g':2,'o':2, 'l':1}
        # - En la iteración 5 donde: letra = e; diccionario = {'g':2,'o':2,'l':1};  diccionario[e] = None; diccionario = {'g':2,'o':2,'l':1,'e':1}
        # - En la iteración 5 donde: letra = .; diccionario = {'g':2,'o':2,'l':1,'e':1};  diccionario[.] = None; diccionario = {'g':2,'o':2,'l':1,'e':1,'.':1}
        # - En la iteración 5 donde: letra = c; diccionario = {'g':2,'o':2,'l':1,'e':1,'.':1};  diccionario[c] = None; diccionario = {'g':2,'o':2,'l':1,'e':1,'.':1,'c':1}
        # - En la iteración 5 donde: letra = o; diccionario = {'g':2,'o':2,'l':1,'e':1,'.':1,'c':1};  diccionario[o] = !None,+=1; diccionario = {'g':2,'o':3,'l':1,'e':1,'.':1,'c':1}
        # - En la iteración 5 donde: letra = m; diccionario = {'g':2,'o':3,'l':1,'e':1,'.':1,'c':1};  diccionario[m] = None; diccionario = {'g':2,'o':3,'l':1,'e':1,'.':1,'c':1,'m':1}
        if(letra in llaves):                    #Si la letra ya está en el JSON existente, le suma un 1 al contador que se encuentra en su value.
            diccionario[letra] += 1
        else:                                   #Si la letra NO está en el JSON existente, la agrega y le asigna un valor de 1.
            #Agregamos letras al diccionario
            diccionario[letra] = 1
    return diccionario                          #El diccionario resultante es: diccionario = {'g':2,'o':3,'l':1,'e':1,'.':1,'c':1,'m':1}.
```

- **Creación de una lista con un bucle for y una función:**

```
#VECTOR X^2 HECHO CON DOS BUCLES FOR Y UNA FUNCIÓN (LISTAS DE COMPRENSIÓN):
```

```
#Función f(x) = x^2
def f(x):
    return x**2
#Las listas en Python pueden ser usadas como vector.
#Declaración de un vector que represente los valores del eje x, de 0 a 10, el último valor del bucle no es
#alcanzado.
x_l_c = [i for i in range (11)]
#Declaración de un vector que represente los valores del eje y, usando los valores del eje x, yendo igual de
#0 a 10 pero elevando dichos valores al cuadrado.
y_l_c = [f(x_l_c[i]) for i in range (11)]

#VECTOR X^2 HECHO CON UNA FUNCIÓN Y UN BUCLE FOR (MÉTODO TRADICIONAL):
#Función f(x) = x^2
def f(x):
    return x**2
#Método tradicional (listas)
x_lista = []
y_lista = []
for i in range(11):
    x_lista.append(i)
    y_lista.append(f(i))
```

- **BUCLE WHILE:** Este es un bucle indeterminado, que se ejecutará un número infinito de veces, hasta que la condición de su paréntesis deje de ser true **while(condición)**.

Objetos y Clases

La POO provee una aproximación para estructurar programas y aplicaciones de tal forma que sus datos y operaciones se agrupen en clases para que se puedan acceder a ellos por medio de objetos, de esta forma en un solo tipo de dato se incluyen ya acciones, características o valores.

Las clases actúan entonces como modelos o moldes que se utilizan para construir varias instancias o ejemplos de objetos similares. Una instancia o un objeto es un ejemplo de una clase. Todas las instancias/objetos de una clase poseen el mismo tipo de variables de datos (características) y acciones que pueden realizar.

Las clases les permiten a los programadores almacenar de forma conceptual información relacionada, lo cual hace que los códigos sean más fáciles de estructurar y de mantener.

La terminología de las clases es:

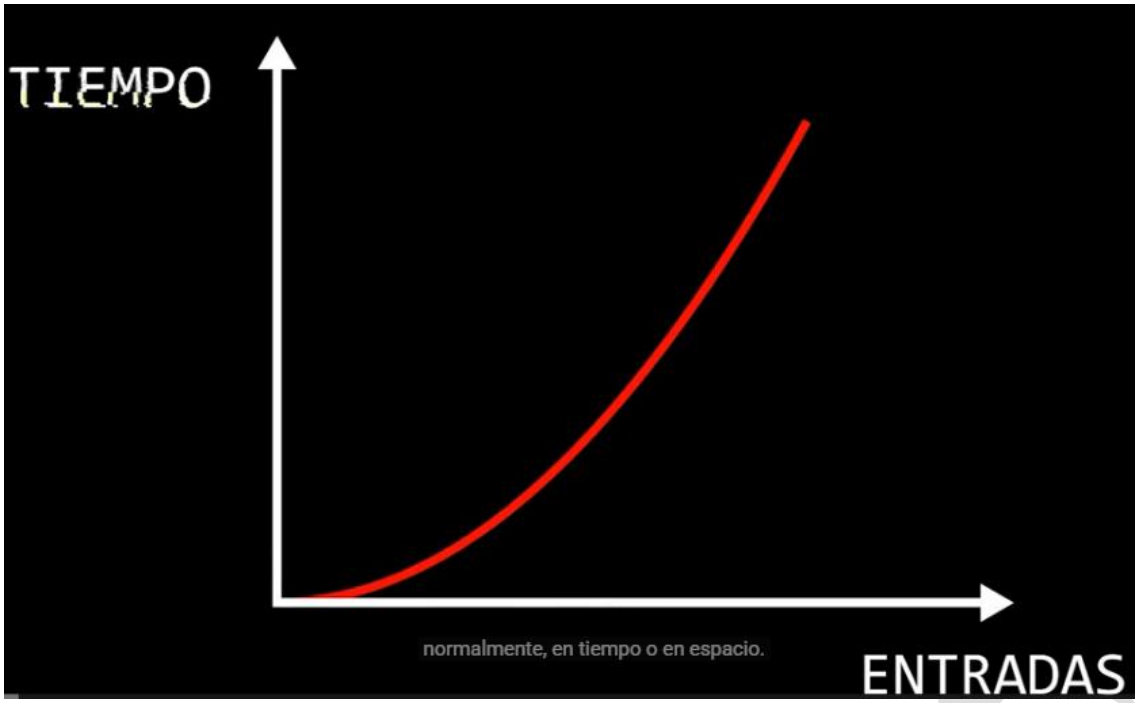
- **Clase:** Define la combinación de datos y comportamientos que operan sobre estos. Una clase actúa como un modelo o molde a través del cual se pueden crear nuevas instancias.
- **Instancias u objetos:** Una instancia también conocida como objeto, es un ejemplo de una clase. Todas las instancias de una clase poseen los mismos datos: *campos (fields)/atributos (attributes)*;

pero un valor propio para cada uno de estos. Cada instancia de una clase responde al mismo conjunto de acciones: *métodos (requests)*.

- **Atributo/Campo/Variable de instancia:** Los datos o variables que posee un objeto se representan por sus atributos. El estado de un objeto en un momento particular se relaciona con los valores corrientes que poseen sus atributos.
- **Método/Función:** Acción que pueden realizar todas las instancias de una clase y se define dentro del paréntesis que describe el parámetro del método al crear un objeto.
- **Interfaz:** Una interfaz define qué métodos debe tener un tipo de clase, pero no cómo se implementan, es un contrato que fuerza a varias clases a implementar como mínimo los mismos métodos, aunque puede agregar sus propios. Su equivalente en Python son las **clases abstractas**.
    - **Clase abstracta:** Es aquella que no puede ser instanciada directamente, sino que otra clase debe heredar de ella para que sus atributos y métodos se apliquen y define uno o más métodos abstractos que deberán ser utilizados forzosamente en las clases que hereden de ellas para garantizar que todas las clases hijas implementen ciertos métodos.

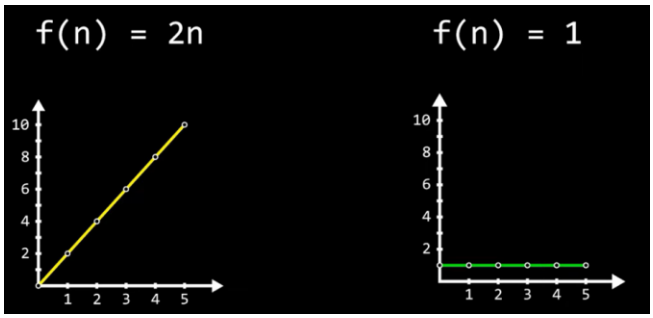
Complejidad Algorítmica

La complejidad de un algoritmo es la **función** que mide cuanto **espacio de memoria** y **tiempo de ejecución** (**recursos**) va a necesitar un algoritmo con **respecto al tamaño de la entrada**.

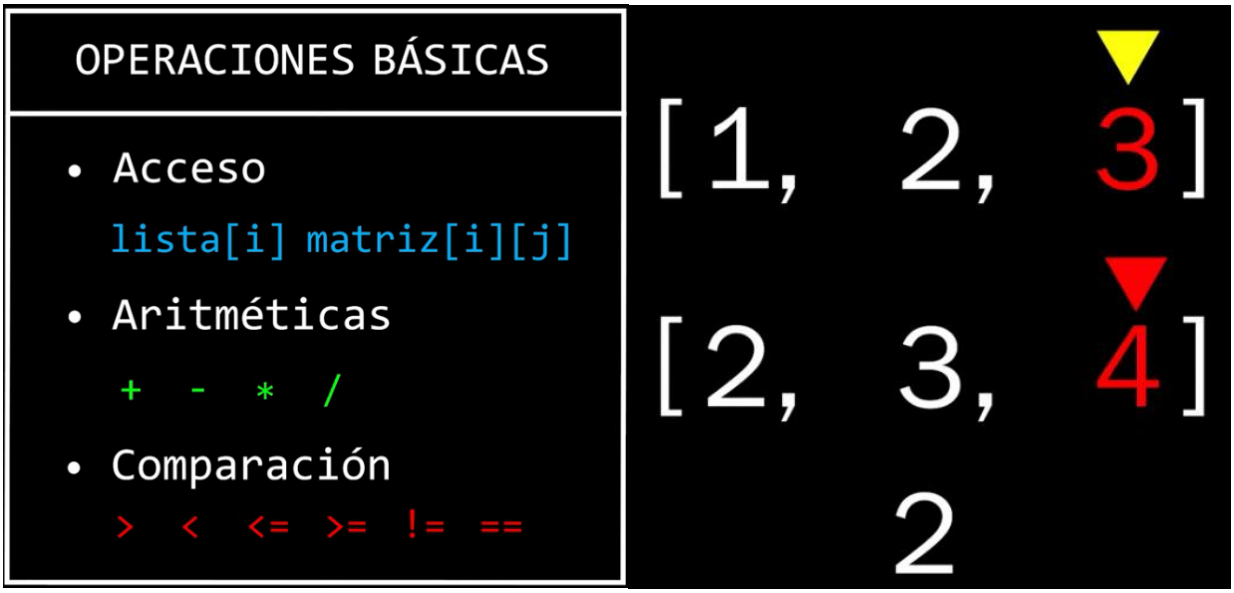


Para medir esto se considera el **número de variables o estructuras de datos auxiliares** al ejecutar nuestro algoritmo y los **bucles o condicionales** que indiquen la estructura del flujo de los datos.

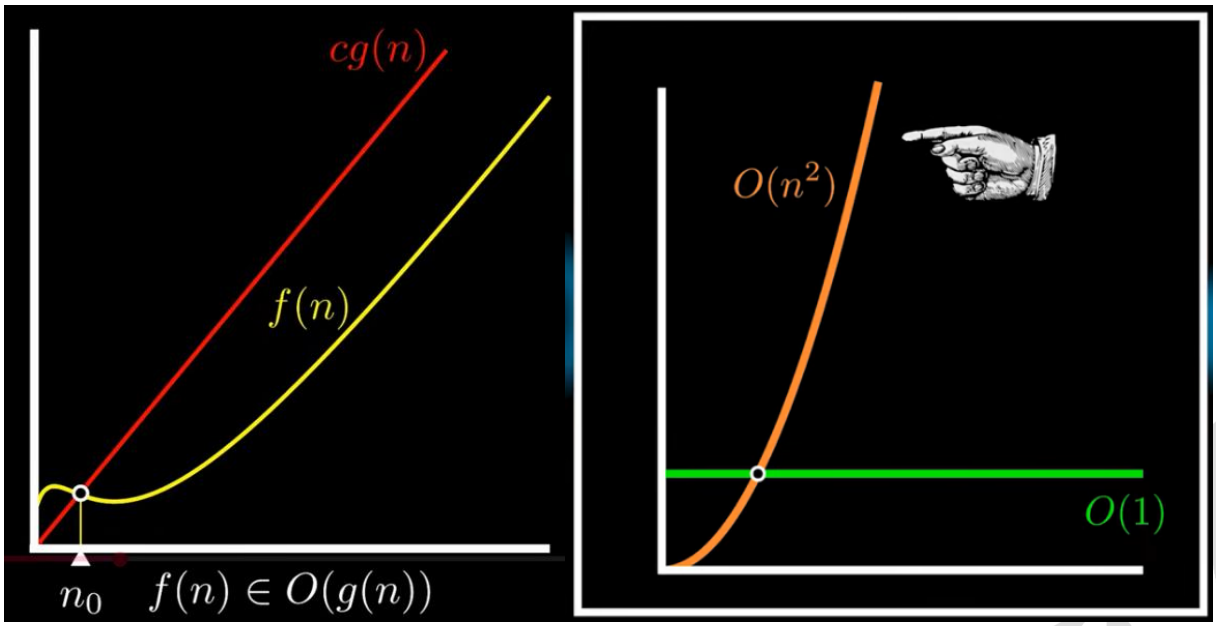
- **Espacio de Memoria:** Por ejemplo, si para reordenar una lista de forma aleatoria, cambiando sus elementos de lugar utilizo dos listas auxiliares, estaría utilizando 2 elementos de memoria durante la ejecución, cuyo tamaño dependerá del número de elementos de la lista de entrada, teniendo así una ejecución de n+n = 2n = O(n), mientras que si para hacer este reordenamiento utilizo una sola variable que accede a cada valor del array, acceda a otra posición random del array, almacene ese valor del elemento en la lista y lo coloque en el índice actual, se está ejecutando la misma acción, pero con el elemento que se encontraba anteriormente en esa posición, así, no importando que tanto crezca la entrada, solamente estoy utilizando 1 sola variable durante la ejecución, teniendo así una complejidad O(1).



- **Tiempo de ejecución:** Para medir el tiempo de ejecución de un algoritmo, se considera las operaciones básicas que realiza como acceso a una variable o los elementos de una estructura de datos, operaciones aritméticas y lógicas (comparación). Una vez teniéndolas en cuenta todas las operaciones que se hacen, se multiplica por el número de elementos de la entrada, obteniendo x_operaciones*n_elementos = xn, pero si aquí, por ejemplo, se utilizan dos o más estructuras de datos en un algoritmo, donde por cada elemento de la primera estructura, se recorre cada elemento de la segunda estructura, se multiplicarían entre ellas, obteniendo $n * n = n^2$ en complejidad, por lo tanto, O(n^2), por eso es peligroso utilizar bucles anidados.

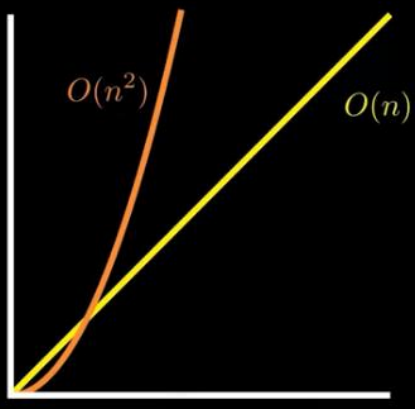


- **Cota Superior Asintótica:** La notación Big O (superior asintótica) indica a que orden de complejidad pertenece nuestra función y siempre corresponderá a la que se refiera al peor caso de ejecución de un algoritmo, ya que las entradas de estos pueden variar, haciendo que su complejidad varíe de igual forma, pero la notación Big O, siempre apuntará a la del peor caso, indicando cuanto como mucho puede crecer la complejidad de ejecución de una función, quitando sus constantes y las de menor orden, quedándonos solo con la variable mayor. Por ejemplo: $O(2n^2 + 3n + 1) = O(n^2)$.
    - **Recursos:** Espacio de memoria y tiempo de ejecución.

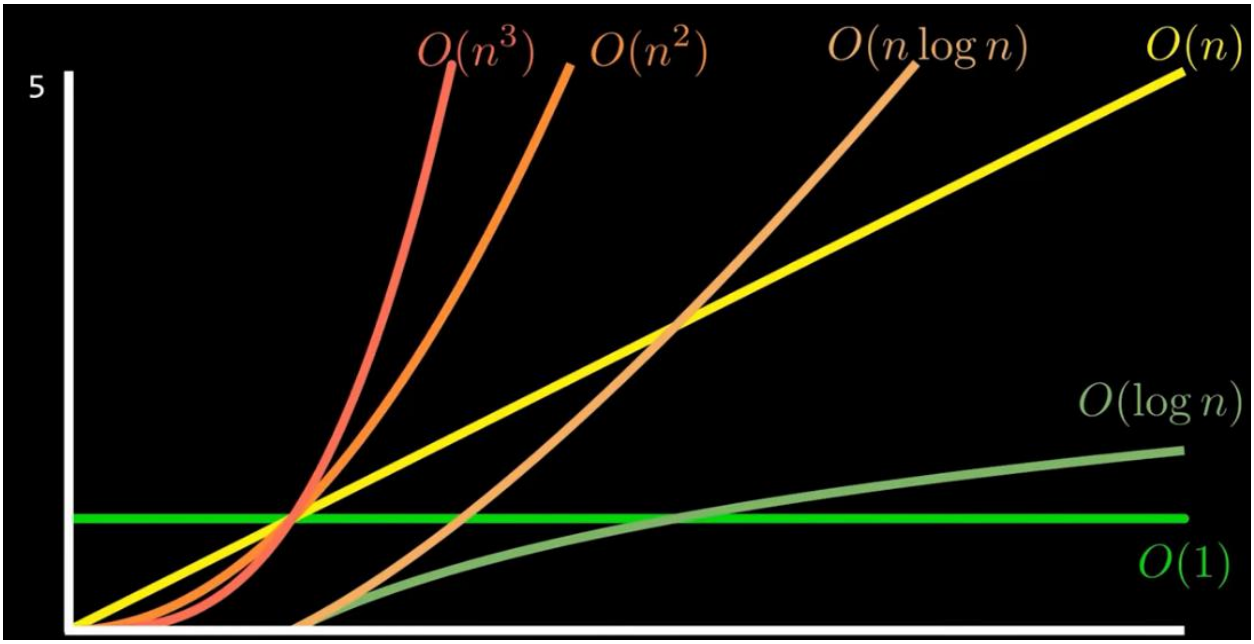


Ahora vemos la importancia de la notación O(n), donde dependiendo de la complejidad del algoritmo, si la entrada es muy grande, el tiempo de ejecución y los recursos necesarios para correrlo, pueden ser insostenibles.

n	O(n)	$O(n^2)$
10	10	100
100	100	10000
1000	1000	1000000



Los diferentes tipos de notaciones en orden son el constante O(1), logarítmico O(log(n)), lineal O(n), cuadrático $O(n^2)$, cúbico, exponencial $O(2^n)$ y el peor es factorial O(n!).



n	O(n)	$O(n^2)$	$O(2^n)$
10	0,001s	0.01s	0,1024s
100	0,01s	1s	$4{,}01 \cdot 10^{18}$ años
1000	0,1s	1m 40s	$3{,}39 \cdot 10^{289}$ años