Notas sobre programación:

La menor unidad de razonamiento lógico que se puede utilizar son el **True** y el **False**. Mediante estos y los operadores relacionales y los mismos lógicos se pueden hacer gran variedad de cosas.

Operador if: mediante este operador logramos que si la expresión que le sigue al if, devuelve **True**, se ejecute el bloque de comando siguiente, si devuelve **False**, entonces o bien no se ejecuta ningún comando, o busca algún comando ejecutable más abajo en un elif o un else.

Operador elif: se usa después de haber usado un operador if, luego de este haber resultado en **False** y no haberse ejecutado, el programa buscara ejecutar la siguiente línea de Código que será un elif, donde, si este resulta en **True**, se ejecutara el código. Se puede tener tantos elif como posibilidades sean necesarias para la situación que se presente. Una vez el if, o uno de los elif sea ejecutados, se saltara los demás elif o el else que se haya colocado por debajo de este, y seguirá al próximo bloque de comando, así que los elif se deben de colocar por orden de prioridad.

Operador else: Este operador se utiliza luego de un if, luego de un elif, o luego de un **while** o un **for** y solo entra en acción cuando el comando anterior devuelve falso. Cuando de utiliza con un comando iterative **while** o **for**, debe de colocarse al mismo nivel que el comando iterativo y no dentro del mismo bloque de comando o este no se ejecutará y devolverá un error.

Operador While: Esto es un operador iterative, lo que significa que cada vez que el bloque de comando posterior devuelva True, ese se seguirá ejecutando, mientras la condición sigue devolviendo True. Normalmente el final del bloque de código del while se debe de colocar una línea de código que, cuando se ejecute, este haga cambiar la primera línea del while, y que en algún punto esta devuelva false, de lo contrario este bloque se ejecutara indefinidamente.

Operador For: Esto es otro operador iterativo, el cual recorre una lista o colección de valores en una lista determinada y solamente itera hasta que haya terminado de recorrer todos los valores de la lista. Ese operador seria lo mismo que decir, por cada valor que hay en la lista (en caso de la lista tener 6 valores seria lo mismo que decir hazlo seis veces) imprime hola. En caso de la lista tener 6 valores este operador iterativo solo sirve para iterar un bloque de comando tantas veces como valores haya en la lista o colección.

Otra forma interesante de usar el for, es con el enumerate.

For índice, valor in enumerate (lista1):

lista [valor] \*10

Esto permitirá, en caso de que sea una lista compuesta por números, multiplicar individualmente cada uno de los valores de dicha lista por 10, sin tener que modificarlo índice a índice, ya que lo que está diciendo el For índice, valor es que no solo me hable de índices, los cuales son solamente números enteros que representan la posición de cada uno de los valores de la lista, sino que me hable de los valores de la lista como tal, permitiéndome modificarlos todos al mismo tiempo.

Por lo que Podemos concluir que el for valor in lista, devuelve un numero entero que corresponde con el índice más grande (esto no coincide con len, ya que los índices comienzan desde el cero, y len comienza desde 1), y además que el for índice, valor in enumerate(lista), devuelve también un numero de índice en caso de no darle una orden de modificación, pero en caso de modificarlo, sí que modifica dicha lista ya que este se refiere a cada valor de la lista con su respective índice.

Tipos de colecciones:

Listas

Conjuntos

Tuplas

Diccionarios

En todas estas colecciones se pueden aplicar los índices y los slicing, excepto en la de diccionario donde en vez de utilizar los índices se utilizan las palabras claves

Las listas y conjuntos son modificables y se les pueden aplicar operadores como .add (en el caso de los conjuntos) o .append (en el caso de las listas)

Para todos los tops de colecciones se puede utilizar el operador .index , que nos dice que el elemento se encuentra en dicho índice, el operador .count, que muestra cuentas veces se muestran ese valor, en caso de ser una lista ya que los conjuntos no permiten que un valor se repita dentro de la misma, el operador sum puede sumar los elementos de una lista conjunto o tupla siempre y cuando el tipo de elemento sea int o float.

Además de esto también Podemos hacer colecciones anidades de colecciones, o sea, tener listas dentro de listas o listas dentro de tuplas, las cuales no pueden ser modificables pero si se pueden mostrar individualmente, así como cada uno de los valores individuales de cada una de las listas anidadas dentro de la lista, conjunto, o tupla principal

En los diccionarios si deseamos agregar entradas nuevas simplemente tenemos que declarar el diccionario [índice a agregar]=valor. Solamente hay que declarar el índice o palabra clave a agregar e igualar lo a un valor que se le quiera dar a dicho índice o palabra clave. En los diccionarios no se puede usar slicing, solamente nos Podemos referir a uno de sus índices o palabras claves.

Para modificar el procedimiento es el mismo, se declara el índice en el diccionario y se iguala al nuevo valor que se le quiera dar.

Para eliminar una entrada no se le da un valor valido en el diccionario porque si no el índice seguiría existiendo , aunque con un valor nulo, sino que se usa del, antes de declarar el índice o palabra clave que se desea eliminar y esta palabra clave será eliminada junto con su respective valor

Como se delimita cada tipo de dato:

Las tuplas están delimitadas por paréntesis. ( ) tupla = (1,2,3)

Las listas por corchetes. [ ] lista = [1,2,3]

Los conjuntos por llaves. { } conjunto = {,1,2,3}

Los diccionarios también por llaves. {1: “uno”} diccionario = {1:”uno”,2:”dos”,3:”tres"}

Las pilas en Python se representan como listas. [ ] pila = [1,2,3]

Las colas se representan como deque ([ ]) cola = deque([1,2,3,])

Especificaciones:

Los diccionarios se diferencian a los conjuntos en que los índices de los conjuntos funcionan como en las listas o en las tuplas o en las colas o en las pilas, con números enteros, pero en los diccionarios es necesario declarar cada uno de los índices con una ‘palabra clave’, que funciona como índice, seguido de dos puntos se encontrara entonces el valor. Entonces para en los diccionarios referirse al índice 0, habrá que poner diccionario (palabra clave que ocupa el lugar del índice) en vez de diccionario [0] como se haría en las demás colecciones de datos.

Luego de los diccionarios le siguen las colas y las pilas. Estos conjuntos son básicamente listas, solo que las utilizaremos en forma de pilas, simplemente usando .append y .pop(). De esta manera solamente se puede agregar elementos al final y sacar elementos del final de la pila, como una pila.

Las colas por otro lado se generan importando un comando deque. Este se puede utilizar para convertir una lista, o pila, en una cola, se representa por llaves y paréntesis y una vez que el tipo de datos es cambiado a deque nos permite utilizar el comando popleft(), con el cual Podemos introducir datos por la derecha, o al final con .append, y a su vez sacar los datos por la izquierda con popleft(), como en una cola real, donde el primer que llega es el primero en salir.

Qué tipo de comandos me permite cada tipo de colección de dato:

Hay dos tipos de funciones:

función( lista ) y lista.función

range: sirve para generar una lista de elementos numéricos del tipo int con tres datos de los cuales solo es necesario el Segundo, que determina hasta donde será de larga la lista range(0,100,2), significa que la lista empieza en cero, termina en el índice 100, o sea el número 99, y el tercer dato es el salto, lo que significa que imprima el cero, e ira 2 números por encima, o sea, este range imprime todos los números pares del cero al 99. Esto puede ser muy útil ya que se puede utilizar para convertir a cualquier tipo de colección de datos posteriormente usando la conversión de datos.

[ ]: Sirve para mostrar el valor de un índice y se puede usar en todos y cada uno de los tipos de datos de todas las colecciones

len (): Sirve para medir la longitud de la colección, o sea, cuantos elementos tiene, este si la colección está vacía, len () devolverá un cero, y si tiene un elemento, devolverá un uno, por el contrario, si se usa .index, nos devolverá un erros, ya que para que .index nos devuelva alguna respuesta, la colección deberá tener al menos un elemento, y, en caso de tener uno, nos devolverá un cero y no un uno, ya que los índices comienzan por cero.

Set: esta funcion es específicamente para crean un conjunto, o cambiar las propiedades de una lista a las

del: permite el comando del, delante de la lista y entre paréntesis, sirve para borrar según el índice y según el slicing.

sum: Sirve para sumar elementos de las colecciones del tipo int o float, mediante slicing. {Se usa en todas las collaciones excepto en el diccionario siempre y cuando el tipo de datos sea int o float}

type: sirve para saber el tipo de datos o colección de estos que estamos tratando, también podemos utilizar el slicing para conocer el tipo de dato que hay dentro de una colección de datos.

def

.append: sirve para suman un elemento al final de la lista, no sirve

.add :En caso de ser un conjunto o un diccionario, que también es un conjunto, este valor no se almacenara al final, sino que se ordenara de manera alfanumérica dentro del conjunto.

.discard: sirve para eliminar un elemento del conjunto al igual que del, la diferencia es que este se pone al final del conjunto al que se le desee aplicar la modificación.

.index: muestra el índice del valor que se esté buscando, en caso de ser un diccionario por supuesto habría que buscar la palabra clave y no el índice int, ya que en los diccionarios los índices se guardan en palabras claves y no en números enteros.

.count: se utiliza para contar cuantas veces existe un valor dentro de una colección, o sea, para ver si el valor esta repetido, o solamente se encuentra una vez, o ninguna

.items( ): esta sería la manera de usar el enumerate en los diccionarios, ya que estos nos permiten utilizar el for, pero no nos permite usar el enumerate. De esta manera, esta función nos devolverá no solo el valor del index, sino que también el index como tal.

Funciones:

Podemos crear nuestras propias funciones definiendo un nombre para la misma , abriendo un paréntesis seguido del nombre para colocar los argumentos a usar en la misma, seguido de dos puntos y debajo se declara el return o la salida que va a dar dicha función.

Luego de que esta función ha sido definida se puede utilizar en un objeto, o en una variable utilizando estos como argumento de esta, y esta devolverá el resultado del código definido en el return:\

def new\_function (argumento1, argumento2, argumento3):

return((argumento1+argumento2+argumento3/3))

#ya aquí la función (new\_function) está definida y la podemos usar.

MEDIA123=new\_function (1,2,3)

print (MEDIA123)

Variables y funciones:

Dentro de una función definida por nosotros podemos o bien llamar una variable antes declarada o podemos declarar una dentro de la función. Aquí es importante saber que ocurren dos procesos completamente diferentes, uno, el proceso principal, y otro el proceso dentro de la función que es como si estuviéramos haciendo otro código completamente diferente por lo que podemos tener dos variables con el mismo nombre, solo que, al llamar a la variable, o a la función, tendremos resultados completamente diferentes según como hagamos esto.

Return:

El uso de return en una función sirve para definir radicalmente que es lo que va a devolver dicha función, y nada por debajo de este se va a ejecutar ya que funciona como un break en un bucle.

Argumentos por valor y referencia:

Por valor:

Una vez que llamamos a la función y utilizamos una variable externa, ósea, generada por el proceso principal, dentro de la función y la llamamos después para ver si la función realizo cambios en esta, nos devolverá la misma función que tenemos en el proceso principal ya que esta no se ve afectada. No es lo mismo declarar una nueva variable que será igual al resultado de pasarle la nueva función a una variable del proceso principal, ya que ahí si mostrara el resultado de lo que hace la función como tal, a lo que nos referimos acá es que una vez que le pasamos una función a una variable del proceso principal, esta no se verá afectada.

**Ejemplo:**

n=10

def doblar\_valor (numero): numero\*=2

doblar\_valor(n) n

*#10*

Por referencia:

Al llamar la función sobre una colección como una lista o un diccionario, sin necesidad que hacer igual una nueva variable del proceso principal al resultado de pasarle la función a la colección, esta se verá modificada en el proceso original. Esto ocurre ya que Python pasa este argumento por referencia por defecto.

colección=[10,20,30]

def doblar\_valor (numero): numero\*=2

for i, n in enumerate (colección): números[i]\*2

doblar\_valor(colección)

*#colección=[20,40,60]*

***Nota:***

Podemos modificar una variable externa utilizando una función sobre ella misma igualando la función sobre la variable a la propia variable, haciendo esto, la variable será igual a la función llamada obre la misma.

EL ZIP:

El zip es otra forma de crear iteraciones y crear listas formadas por tuplas de elementos de otras colecciones separados por comas. En este se puede hacer unos de el for in, ya que es un objeto iterable, una lista, solo que en esta, el primer elemento que la forma, será una tupla formada por el primer elemento de la primera lista, coma, el primer elemento de la segunda lista, coma, el primer elemento de la tercera lista…unidos todos en una tupla, que funciona como primer elemento de la nueva lista creada con zip, y así sucesivamente con los elementos siguientes.

El zip seguirá añadiendo elementos a la nueva lista, mientras no se agote una de las colecciones de las que recibe los datos para formar el zip.

Lo curioso del zip, es que se puede mas adelante descomprimir y acceder a cada uno de sus elementos individualmente haciendo unos de \*zip(suponiendo que nuestro zip se llame zip), y esto nos devolverá una lista a, formada por los primeros elementos de cada una de las tuplas, una lista b, formada por los segundos elementos de cada una de las tuplas, una lista c, formada por los terceros elementos de cada una de las tuplas, y así sucesivamente, hasta que se agote el zip.

EJ:

a=[1,2,3] d=[(1,’uno’,one),(2,’dos’,’two’),(3,’tres’,’three’)]

b=[‘uno’,’dos’,’tres’] zip(\*d)

c=[‘ine’,’two’,’three’] print(a)

d=zip(a,b,c) a=[1,2,3]

El Formateo de Str:

Las cadenas de textos, o sea, los valores encerrados entre comillas simples o dobles, tiene una fusión llamada .format la cual es bien especial, ya que nos permite obviar el uso de comas para poner una variable y llamarla en el print, en vez de esto podemos tomar la cadena y llamar la variable directamente encerrada entre llaves y cuando cerramos las comillas llamar a la función .format con vas variable que vamos a llamar de forma ordenada y separada por comas, o aún mucho más fácil podemos poner la variable dentro de las llaves, lo que seria una forma mucho mas intuitiva .de usar el .format.

Ejemplo:

b=”un texto entre llaves”

c=7

print(“imprimir”, b, c)

*## imprimir un texto entre llaves 7*

Con format:

print (“imprimir {} {}”.format(b, c)

*## imprimir un texto entre llaves 7*

print (“imprimir {0} {1}”.format(b, c)

*## imprimir un texto entre llaves 7*

En este caso estamos tomando el interior del format como si fuera una lista y a lo que nos estamos refiriendo es al índice de la lista, y como esta tiene un solo elemento nos referimos al índice cero.

En caso de que esta lista format tenga mas de un elemento podemos cambiar el orden en que se muestran cada uno de los elementos entre llaves con solo cambiar el índice dentro de la llave, y se mostrará así el valor correspondiente a dicho índice.

Ejemplo:

print (“imprimir {1} {0}”.format(b, c)

*## 7 imprimir un texto entre llaves*

***ARGS & KWARGS:***

A la hora de crear funciones siempre debemos de introducir como mínimo un argumento para poder utilizar la misma, en algunos casos, necesitamos más de uno, ¿pero ¿qué pasa cuando necesitamos una cantidad de argumentos que desconocemos?

La respuesta a esto son los ARGS y los KWARGS, que no son más que argumentos indeterminados que le pasamos a una función, que difieren en la manera de acceder a ellos y en el tipo de argumento que se usan.

Se usan de la siguiente forma:

def funcion\_indeterminada (\*args) Aquí le hemos dicho a la función que tome indeterminados argumentos y que con ellos haga lo que le ordenemos a continuación. Los args se almacenan en una tupla.

def funcion\_indeterminada (\*\*kwargs) Aquí le hemos dicho a la función que tome indeterminados argumentos y que con ellos haga lo que le ordenemos a continuación, con la diferencia de que a la hora de introducir estor=s argumentos los introduciremos con una clave y con un valor, ya que los kwargs, son diccionarios.

**Combinación de ARGS y KWARGS:**

Podemos combinas los valores normales, o argumentos esperados en una función, con más argumentos del tipo args o kwargs, siempre que introduzcamos los argumentos en el orden correcto, siendo este orden, valores esperados, valores args, colecciones kwargs. Luego podemos trabajar con estos independientemente, o u en combinaciones los unos con los otros siempre que lo permitan las operaciones entre cada uno de los tipos de datos