Análisis de datos

Nivel Básico – Explorador

Misión 1

Estructura de Datos

Listas

Una lista es una estructura de datos ordenada, lo cual se puede definir como un tipo de dato conformado por múltiples datos (que pueden ser de diferentes tipos de datos). Las listas suelen interpretarse como una fila de casillas, donde en cada casilla se guarda un dato que se haya ingresado; a cada casilla le corresponde un número de identificación, denominado *subíndice*.

'a'	'c'	'Hello'	18	True
0	**1**	**2**	**3**	**4**

a. Creación de una lista

Una lista se crea llamando los datos que harán parte de la lista, separarlos con comas y encerrarlos todo entre corchetes. Si se desea asignar una lista a una variable, basta con asignar todo lo previo a un identificador de variable.

```
my_list = ['a', 'c', 'Hello', 18, True]
print(my_list)
```

Note que no aparecen los signos ">>>", lo cual significa que no se usa el *interprete interactivo*. El código escrito está en un archivo fuente, así que se debe ejecutar y se mostrará una salida por consola.

Salida por consola:

```
['a', 'c', 'Hello', 18, True]
```

b. Acceso a una lista

Los datos dentro de una lista son ordenados. Esto significa que cada uno está asociado a un número en la lista llamado *subíndice*, el cual determina su posición. Es preciso aclarar que, a diferencia de como se suele hacer en la vida cotidiana, en programación se suele contar desde el número **0** en vez del número **1**, siendo así 0 el subíndice del primer dato, 1 el subíndice del segundo dato y así sucesivamente hasta el número ***n – 1***, donde ***n*** corresponde a la cantidad de datos que contiene la lista.

Para acceder al dato específico de una lista se necesita conocer el subíndice correspondiente a este; se escribe el identificador de una lista seguido de corchetes entre los cuales se escribe el subíndice del dato deseado.

Continuando con el ejemplo anterior...

```
print(my_list[2])
```

Salida por consola:

```
'Hello'
```

Se puede utilizar un subíndice negativo, el cual servirá para acceder a la lista en orden opuesto, desde el último hasta el primero, donde **–**1 corresponde al último dato, **–**2 al penúltimo y así sucesivamente hasta **–*n***, donde ***n*** corresponde a la cantidad de datos que contiene la lista.

```
print(my_list[-2])
18
```

c. Secciones de listas

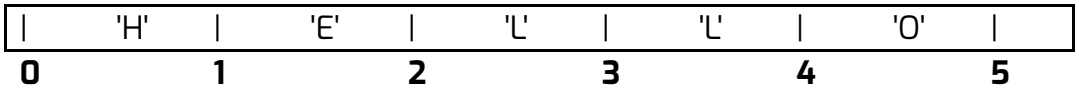
Una utilidad presente en las listas es la de poder crear “rebanadas” o secciones de lista a partir de una lista existente, creando así *sublistas*. La sintaxis para esta tarea es similar a la usada para acceder a un dato, pero en vez de un subíndice se escriben dos números separados por *dos puntos* “:”. El primer subíndice corresponde al primer dato que hará parte de la nueva lista, y el segundo subíndice corresponde al subíndice del dato siguiente al último dato que hará parte de la lista.

```
my_list = ['H', 'e', 'l', 'l', 'o']
 print(my_list[1:3])
```

Salida por consola:

```
['e', 'l']
```

Una manera sencilla de verlo es imaginarse nuevamente los datos encerrados entre casillas, y los subíndices ahora serán las barras laterales que separan los datos (los extremos también tienen barras). Entonces, los subíndices de "secciones de lista" corresponden a las barras que encierran todos los datos que harán parte de la nueva lista.



Así se hace más claro que al tomar la sección correspondiente a [1:3] se seleccionan los datos entre las barras 1 y 3, los cuales son 'E' y 'L'.

Si se omite alguno de los subíndices, la nueva lista contendrá los datos hasta el extremo en la dirección del dato que se omitió. Si se omiten ambos subíndices, la nueva lista será una copia exacta de la original.

```
print(my_list[1:])    # Se omite el subíndice derecho
print(my_list[:4])    # Se omite el subíndice izquierdo
print(my_list[:])     # Se omiten ambos subíndices
```

Salida por consola:

```
['e', 'l', 'l', 'o']
['H', 'e', 'l', 'l']
['H', 'e', 'l', 'l', 'o']
```

d. Modificar datos de una lista

Para modificar el dato en un subíndice específico de una lista, simplemente se asigna el nuevo dato al "acceso del dato", es decir, se usa la misma sintaxis para acceder a un dato seguido de un signo de asignación "=" y el nuevo valor, de la misma manera que se haría con una variable convencional.

```
my_list = [0, 1, 2, 3, 4]
print(my_list)
my_list[0] = 5
```

```
print(my_list)
```

Salida por consola:

```
[0, 1, 2, 3, 4]
[5, 1, 2, 3, 4]
```

Si se realiza la misma modificación, pero en vez de modificar un solo dato en la lista se modifican múltiples datos usando la sintaxis de "secciones de lista" y asignando una lista, se reemplazan todos los datos en esa sección por los datos en la lista asignada; así, si la cantidad de datos en la lista asignada es mayor que la de la sección de lista, la lista se expandirá, y si la cantidad de datos en la lista asignada es menor, la lista se reducirá.

```
my_list = [5, 1, 2, 3, 4]
my_list[1:] = [6, 7]
 print(my_list)
```

Salida por consola:

```
[5, 6, 7]
```

Note que la cantidad de datos en la lista se redujo debido a que la lista que se asignó tenía menor cantidad de datos que la sección que se seleccionó.

e. Agregar datos a una lista

Las listas son un tipo de dato extensible. Esto significa que permiten agregar datos en cualquier momento y existen varias formas en las que se puede concretar esta tarea. Anteriormente se usaron algunas *funciones* (print(), input(), str(), etc...), aunque no se haya explicado su origen y uso; ahora se usarán los *métodos*, los cuales corresponden a una *función* que se aplica sobre una variable.

La primera forma para agregar datos en una lista es utilizando el *método* ***append()***, lo cual se hace escribiendo el *indicador* que corresponde a la lista objetivo, seguido de un punto y el llamado del *método.* Entre paréntesis se escribe el dato que se desea agregar. Esto agregará el dato al final de la lista.

```
my_list = [1, 2]
```

```
my_list.append(3)
print(my_list)
```

Salida por consola:

```
[1, 2, 3]
```

Si se desea agregar múltiples datos, se debe crear una lista con los datos a agregar y usando el *método* ***extend()*** se envía la lista entre los paréntesis del método. Esto agregará los datos al final de la lista.

```
my_list = [1, 2]
my_list.extend([3, 4])
print(my_list)
```

Salida por consola:

```
[1, 2, 3, 4]
```

Una manera de obtener el mismo resultado sin el uso de *métodos* es realizando una "suma" entre listas.

```
my_list = [1, 2]
my_list += [2, 4]
print(my_list)
```

Salida por consola:

```
[1, 2, 3, 4]
```

Finalmente, se presentan dos formas para añadir datos en medio de una lista. La primer forma consiste en utilizar el *método* ***insert()***, al cual se le envían entre paréntesis dos datos separados por comas: el primer dato corresponde al subíndice de la lista a partir del cual se van a mover los otros datos para insertar el nuevo dato; el segundo dato es el nuevo dato que se va a insertar.

```
my_list = [1, 2, 3]
my_list.insert(1, 4)
print(my_list)
```

Salida por consola:

```
[1, 4, 2, 3]
```

Note que a partir del subíndice 1 se movieron todos los datos un espacio a la derecha para ingresar el dato 4 en esa posición.

La segunda forma de obtener el mismo resultado es usando la modificación de "secciones", pero se debe insertar una lista de datos (puede ser de un solo dato) y los subíndices serán el mismo número, el cual corresponderá a la posición en la que se desea insertar.

```
my_list = [1, 2, 3, 4]
my_list[1:1] = [5, 6]
 print(my_list)
```

Salida por consola:

```
[1, 5, 6, 4, 2, 3]
```

Note que todos los datos a partir del subíndice 1 se movieron dos espacios a la derecha para insertar la nueva lista.

f. Eliminar datos de una lista

Las listas deben ser continuas, lo cual significa que no deben existir espacios vacíos entre los datos; los espacios vacíos solo se podrían crear tras eliminar un dato, pero Python moverá todos los datos que están a la derecha para llenar el vacío. Con eso aclarado, se pueden explicar los métodos para eliminar datos en una lista.

El primer método que se presentará consiste en usar la palabra clave ***del*** seguida del acceso al dato que se desea eliminar en la lista.

```
my_list = [1, 2, 3]
 del my_list[1]
 print(my_list)
```

Salida por consola:

```
[1, 3]
```

Los siguientes métodos son exactamente eso, *métodos*. El primer *método* es ***remove()***, en el cual entre paréntesis se envía el ***dato exacto*** (no el subíndice) que se desea eliminar de la lista.

```
my_list = ['H', 'e', 'l', 'l', 'o']
 my_list.remove('o')
```

```
print(my_list)
```

Salida por consola:

```
['H', 'e', 'l', 'l']
```

El segundo *método* es ***pop()***. Entre paréntesis se envía el subíndice del dato que se quiere eliminar y este devolverá el dato que se eliminó.

```
my_list = ['H', 'e', 'l', 'l', 'o']
print(my_list.pop(1))
print(my_list)
```

Salida por consola:

```
'e'
['H', 'l', 'l', 'o']
```

Note que primero se imprime el dato que se eliminó y luego se imprime la lista sin ese dato.

La última manera que se presentará para eliminar datos de una lista consiste en volver a utilizar la asignación por "secciones", donde se escoge la sección deseada a eliminar y se le asigna una lista vacía. Esto hará que todos los datos que hacen parte de la rebanada sean reemplazados por una lista sin datos, eliminado así los datos de la lista.

```
my_list = ['H', 'e', 'l', 'l', 'o']
my_list[1:4] = []
print(my_list)
```

Salida por consola:

```
['H', 'o']
```

Tuplas

Las tuplas son una estructura de datos ordenada, muy similares a las listas, también con un subíndice asignado a cada dato, pero la única diferencia que lo caracteriza es su *inmutabilidad*, lo cual causa que no se pueda modificar la tupla desde su creación, es decir, no se pueden agregar, eliminar ni reasignar datos de la tupla.

a. Creación de una tupla

Las tuplas se pueden crear simplemente escribiendo una secuencia de datos separados por comas; no obstante, por convención se suelen usar los paréntesis para agrupar los datos que harán parte de la tupla.

```
my_tuple = (1, 2, 3, 4, 5)
 print(my_tuple)
```

Salida por consola:

```
(1, 2, 3, 4, 5)
```

Existe un pequeño problema cuando se desea crear una tupla de un solo dato, ya que Python solo reconocerá que es un dato individual. La solución es escribir el dato seguido de una coma, y así se reconocerá esto como una secuencia de datos de un solo dato.

```
my_tuple = (1, )
 print(my_tuple)
```

Salida por consola:

```
(1,)
```

b. Acceso a una tupla

Para acceder a los datos de una tupla se realiza el mismo procedimiento que con una lista: se escribe el indicador de la tupla seguido del subíndice del dato al que se desea acceder, encerrado entre corchetes. Hay que recordar que también se puede usar un subíndice negativo.

```
my_tuple = (1, 2, 3, 4, 5)
 print(my_tuple[3])
```

Salida por consola:

```
3
```

c. Secciones de tuplas

Al igual que en las listas, se pueden tomar secciones de tuplas de una tupla existente. La sintaxis es la misma: se escribe el indicador de la tupla seguido de los subíndices de los datos que harán parte de la nueva tupla encerrados entre corchetes y separados por dos puntos. Hay que

recordar la interpretación de las barras de casilla para entender bien el proceso de secciones.

```
my_tuple = (1, 2, 3, 4, 5)
 print(my_tuple[2:5])
```

Salida por consola:

```
(3, 4, 5)
```

d. Modificar datos de una tupla

Ya dicho, las listas son *inmutables,* los datos que lo conforman no se pueden reasignar.

e. Agregar datos en una tupla

Nuevamente, debido a la *inmutabilidad* de las tuplas, no es posible agregar nuevos datos a la MISMA tupla. Pero dado el caso de que se desee tener todos los datos de la tupla con nuevos datos se puede reasignar la tupla, pero sumando otra tupla con los datos que se desean agregar.

```
my_tuple = (1, 2, 3)
 my_tuple += (4, 5)
 print(my_tuple)
```

Hay que recordar que, al usar un operador de asignación, no se modifica el dato de la variable, sino que se le asigna un nuevo valor operado. Esto significa que, en el ejemplo, se está asignando la tupla sumada la nueva tupla.

Salida por consola:

```
(1, 2, 3, 4, 5)
```

f. Eliminar datos de una tupla

Finalmente, los datos en una tupla no se pueden eliminar debido a que las tuplas son *inmutables*.

Sets

Los sets son la estructura de datos que representan los *conjuntos matemáticos*. Los sets presentan varias características particulares en comparación con las estructuras de datos vistas: son desordenadas, es decir, no se mantiene registro alguno del orden en el que se ingresan los datos, entonces no existe un subíndice el cual se asigna a cada dato. Los datos de un set son únicos, es decir, no se pueden tener múltiples datos iguales. Finalmente, los sets permiten realizar operaciones de conjuntos entre ellos.

a. Creación de un set

Los sets se crean al encerrar entre llaves "{ }" una secuencia de datos separados por comas.

```
my_set = {5, 8, 2, 9, 4}
print(my_set)
```

Salida por consola:

```
{2, 4, 5, 8, 9}
```

Note que el orden de los datos no es el mismo al orden en el que se ingresaron los datos.

En el caso particular de que se desee crear un set vacío, no basta con asignar llaves vacías, ya que esto crearía otro tipo de estructura de datos que se verá más adelante. Así que para crear un set vacío se tiene que asignar la *función* ***set()***, la cual devolverá un set vacío.

```
my_set = set()
print(my_set)
```

Salida por consola:

```
set()
```

Note que se imprime la función que crea el dato en vez de llaves vacías, ya que esto representaría otro tipo de estructura de datos.

b. Acceso a un set

Como ya se dijo, los datos de un set son desordenados, así que no hay una manera de acceder a un dato específico del set. Pero se pueden evaluar todos los datos uno por uno usando un iterador, lo cual se explicará más adelante en el módulo.

c. Operar sets

Existen varias operaciones especiales que se pueden realizar entre los sets, operaciones que corresponden a las realizadas en conjuntos matemáticos convencionales.

i. Contención

La primera operación se basa en evaluar si un dato se halla contenido dentro de un set (esta operación en realidad se puede utilizar con cualquier otra estructura de datos). Así bien, se puede saber si un dato pertenece o no pertenece a una estructura de datos usando la *palabra clave* ***in*** (usada previamente junto al iterador ***for***), la cual se escribe después del dato que se quiere evaluar y antes del identificador de la estructura en la que se quiere evaluar el dato. Esto devolverá como resultado un booleano.

```
my_set = {'S', 'E', 'T'}
 print('S' in my_set)
```

Salida por consola:

```
True
```

ii. Unión

La operación correspondiente a la *unión* es un *método* llamado ***union()***, el cual se aplica sobre uno de los sets sobre los que se aplicará la operación y se envía entre paréntesis el otro set. Este *método* devolverá un set que contiene TODOS los datos de ambos sets.

```
set_1 = {1, 2, 3}
 set_2 = {2, 4, 6}
 union_set = set_1.union(set_2)
 print(union_set)
```

Salida por consola:

```
{1, 2, 3, 4, 6}
```

Note que no se repite el número 2 ya que los datos de un conjunto son únicos.

iii. Intersección

La operación correspondiente a la *intersección* es un método llamado ***intersection()***, el cual se llama sobre uno de los sets sobre los que se aplicará la operación y se envía entre paréntesis el otro set. Este *método* devolverá un set que contiene los datos que pertenecen a ambos sets A LA MISMA VEZ.

```
set_1 = {1, 2, 3}
 set_2 = {1, 3, 5}
 intersection_set = set_1.intersection(set_2)
 print(intersection_set)
```

Salida por consola:

```
{1, 3}
```

Note que los datos 1 y 3 pertenecen a ambos sets.

iv. Diferencia

La operación *diferencia* se efectúa usando el *método* ***difference()***, el cual devuelve un set con los datos que pertenecen al primer set pero NO pertenecen al segundo set. Este *método* se aplica sobre el primer set y entre paréntesis se envía el segundo set.

```
set_1 = {1, 2, 3, 4, 5}
 set_2 = {1, 3, 5}
 difference_set = set_1.difference(set_2)
 print(difference_set)
```

Salida por consola:

```
{2, 4}
```

v. Diferencia simétrica

La operación *diferencia simétrica* se efectúa usando el método ***symmetric_difference()***, el cual se aplica sobre uno de los sets a operar y se envía entre paréntesis el otro set. Este método devuelve como resultado TODOS los datos de ambos sets, pero excluyendo los datos que hacen parte de ambos sets A LA MISMA VEZ. En términos de otras operaciones, la *diferencia* entre la *unión* y la *intersección*.

```
set_1 = {2, 4, 6, 8, 10}
 set_2 = {4, 8, 12, 14}
 symmetric_difference_set = set_1.symmetric_difference(set_2)
 print(symmetric_difference_set)
```

Salida por consola:

```
{2, 6, 10, 12, 14}
```

d. Modificar datos de un set

Debido a que no hay una manera de acceder a un dato específico de un set, estos no se pueden reasignar.

e. Agregar datos a un set

En un set solo se pueden agregar datos usando uno de dos *métodos*. El primero es ***add()***, al cual se le envía entre paréntesis el dato que se desea agregar.

```
my_set = {5, 2, 8}
 my_set.add(9)
 print(my_set)
```

Salida por consola:

```
{8, 9, 2, 5}
```

El segundo *método* se llama ***update()***, y este permite agregar múltiples datos a la vez enviando entre paréntesis una estructura de datos con los datos que se desean agregar. Además, se pueden enviar múltiples estructuras de datos separadas por comas.

```
my_set = {3, 1}
 my_set.update([2, 4], {6, 7})
 print(my_set)
```

Salida por consola:

```
{1, 2, 3, 4, 6, 7}
```

f. Eliminar datos de un set

Para eliminar datos de un set se puede usar uno de tres *métodos.* El primero es el *método* ***remove()***, al cual se le envía entre paréntesis

el **dato exacto** que se desea remover de la lista, es decir, se necesita conocer el dato que se desea eliminar.

```
my_set = {'S', 'E', 'T'}
 my_set.remove('S')
 print(my_set)
```

Salida por consola:

```
{'E', 'T'}
```

El segundo *método* es el anteriormente visto ***pop()***, el cual elimina un dato de la estructura de datos y además lo devuelve. Lo único que se debe tener en cuenta es que, al ser datos desordenados, se eliminará un dato de manera arbitraria.

```
my_set = {'S', 'E', 'T'}
 print(my_set.pop())
 print(my_set)
```

Salida por consola:

```
{'E', 'T'}
```

Finalmente, el tercer *método* para eliminar datos de un set es ***clear()***, el cual no recibe ningún dato entre paréntesis y su función es la de eliminar todos los datos en el set.

```
my_set = {1, 2, 3, 4, 5}
 my_set.clear()
 print(my_set)
```

Salida por consola:

```
set()
```

Diccionarios

Los *diccionarios* en Python se pueden definir como una lista desordenada de pares de datos, donde el primer dato del par se conoce como *llave* y funciona de la misma manera en la que un subíndice lo haría, ya que permite acceder al segundo dato, el cual corresponde al dato que se desea almacenar con dicha *llave*.

a. Creación de un diccionario

La razón por la que anteriormente no se podía crear un set vacío simplemente usando llaves es porque se crearía un *diccionario*. Para crear un *diccionario* se deben encerrar entre *llaves* una secuencia de pares de datos separados por comas y los datos dentro de los pares son separados por dos puntos ":". La llave puede ser casi cualquier tipo de dato, pero se suelen utilizar strings.

```
my_dict = {'name': 'Frank', 'age': 18}
 print(my_dict)
```

Salida por consola:

```
{'name': 'Frank', 'age': 18}
```

b. Acceso a un diccionario

Como se mencionó anteriormente, las llaves en un *diccionario* funcionan en forma similar a un subíndice, de tal manera que se accede a un dato a través del uso de su llave como un subíndice entre los corchetes. Se escribe el indicador del *diccionario* seguido de la llave relacionada al dato al cual se desea acceder encerrado entre corchetes.

```
my_dict = {'name': 'Frank', 'age': 18}
 print(my_dict['age'])
```

Salida por consola:

```
18
```

c. Modificar datos de un diccionario

Si se desea modificar el dato de un *diccionario* es necesario tener la llave de este, y así acceder al dato por medio de la llave y a este se le asigna el nuevo dato deseado. Se escribe el *indicador* del *diccionario* seguido de la llave relacionada al dato encerrado entre corchetes; luego, con el operador asignar "=" se escribe el dato que se desea asignar al dato correspondiente a la llave.

```
my_dict = {'name': 'Frank', 'age': 18}
 my_dict['age'] += 1;
 print(my_dict)
```

Salida por consola:

```
{'name': 'Frank', 'age': 19}
```

d. Agregar datos a un diccionario

El proceso de agregar pares de datos a un *diccionario* es similar al de modificar datos, ya que se necesita asignar un valor a un acceso con una llave, solo que, en este caso, la llave que se utilice debe ser una llave no existente en el diccionario.

```
my_dict = {'name': 'Frank', 'age': 18}
 my_dict['nick'] = 'Frany'
 print(my_dict)
```

Salida por consola:

```
{'name': 'Frank', 'age': 19, 'nick': 'Frany'}
```

e. Eliminar datos de un diccionario

La manera más directa para eliminar datos de un *diccionario* es con el uso de la palabra clave ***del*** seguida del acceso al dato que se desea eliminar.

```
my_dict = {'name': 'Frank', 'age': 18}
 del my_dict['age']
 print(my_dict)
```

Salida por consola:

```
{'name': 'Frank'}
```

Y la otra forma es usando el *método* ***pop()***, al cual se le envía entre paréntesis la llave del dato que se desea eliminar del diccionario; además, devuelve el dato correspondiente a la llave ingresada después de eliminarlo de la lista.

```
my_dict = {'name': 'Frank', 'age': 18}
 print(my_dict.pop('age'))
 print(my_dict)
```

Salida por consola:

```
18
{'name': 'Frank'}
```

Iteración en estructuras de datos

Anteriormente se dio una explicación muy por encima del funcionamiento del iterador ***for***. La razón de esto es que este iterador funciona de manera particular en Python, ya que su verdadero propósito es el de recorrer estructuras de datos.

a. Iterador for en otros lenguajes de programación

En otros lenguajes de programación el iterador ***for*** ejecuta un bloque de código al tomar una variable para contar y darle un valor inicial, luego se define una condición para salir del ciclo (igual que en un ***while***, por ejemplo, que la variable sea menor que un valor límite), y finalmente se define la razón de cambio de la variable iteradora, es decir, cómo se va a modificar la variable en cada iteración. Un ejemplo básico, y el más usual, es iniciar una variable ***i*** en 0, luego definir la condición de iteración como ***i < n***, donde n es el número de veces que se desea iterar. Finalmente se define la razón de cambio como ***i += 1***, y en cada iteración el contador aumenta en uno.

Sin embargo, este iterador se puede recrear en Python utilizando un iterador ***while***, creando la variable previamente y dar la razón de cambio dentro del bloque a iterar.

```
i = 0              # Inicializar Variable
 while (i < 3):       # Definir Condición
print(i)         # Sentencias
i += 1           # Definir Razón de Cambio
```

Salida por consola:

```
0
1
2
```

Con esto en mente, se puede utilizar este método de iteración para recorrer estructuras de datos ordenadas.

```
my_list =                                          [2, 4, 8]
 i = 0
 while (i < 3):
print(my_list[i])
i += 1
```

Salida por consola:

```
2
4
8
```

b. Iterador for en Python

Distinguiendo que en el módulo de ***control de flujo*** se busca explicar el iterador ***for*** basado en su funcionamiento usual en otros lenguajes de programación, se puede entrar a profundizar el verdadero funcionamiento de este iterador en el lenguaje de programación Python.

El iterador ***for*** de Python es el equivalente a un iterador ***foreach*** de otros lenguajes de programación, el cual consiste en recorrer una estructura de datos, y en cada iteración a la variable definida se le asigna un elemento diferente; así bien, si es una estructura de datos ordenada, simplemente se recorre en orden, pero si es una estructura de datos desordenada, en la cual habría problemas por su poca accesibilidad recorriéndola de otras maneras, el ***for*** de Python permitirá recorrer esta sin ningún problema ya que al final se habrá accedido a todos los elementos.

```
my_set = (3, 1, 5, 4)
 for i in my_set:
print(i)
```

Salida por consola:

```
3
1
5
4
```