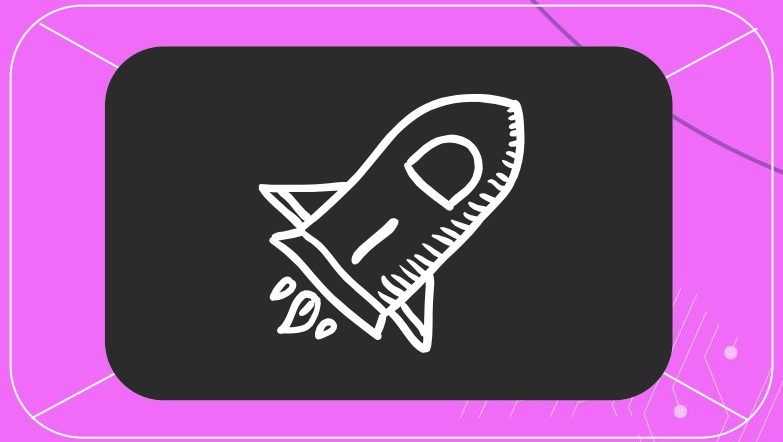
Gobierno del CHACO | Ministerio de la Producción y el Desarrollo Económico Sostenible

INFORMATORIO

Semana 3
¡Bienvenidos!



Temas de hoy

1. Estructuras de datos: Listas, Tuplas, Conjuntos, Diccionarios.
2. Métodos y operaciones comunes con estructuras de datos.

1

Estructuras de datos

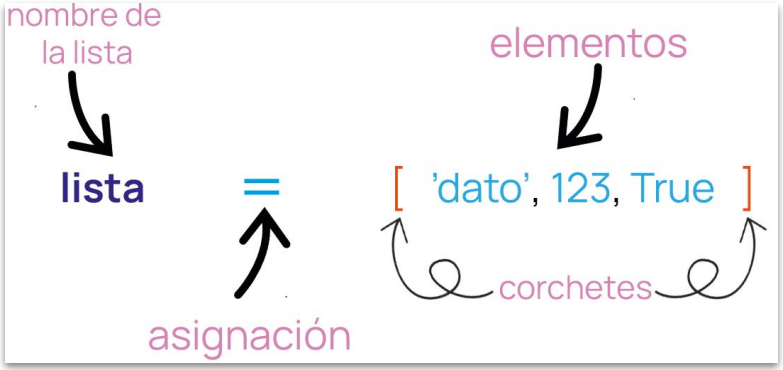
Un problema que tenemos con las variables es que no almacenan varios tipos de datos a la vez, por lo que si necesitamos guardar un registro de varios datos, no lo podríamos lograr.

Para solucionar esto existen las estructuras de datos, las cuales nos permitirán organizar y almacenar información de manera eficiente 👉

Estructuras de datos: Lista - list

Listas en Python

Esta estructura la podemos encontrar en la mayoría de los lenguajes de programación con distinto nombre, como “arreglo”, “vector”, “secuencia”, “colección”, entre otros, pero sigue haciendo básicamente lo mismo, almacenar y organizar distintos tipos de datos a la vez en una sola estructura.
Específicamente las listas son colecciones **ordenadas** y **mutables** de elementos, donde cada elemento puede ser de cualquier tipo de dato (números, cadenas de caracteres, booleanos, etc). Son ideales para almacenar y gestionar conjuntos de datos relacionados.
Su estructura básica es muy similar a la de una variable con algunas diferencias:



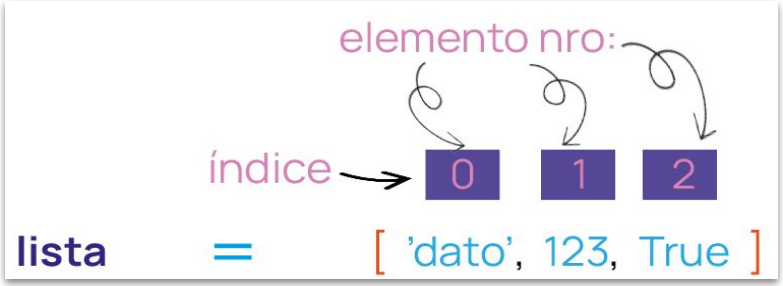
En términos generales, Python diferencia estas estructuras por el símbolo que encierra la información. Para el caso de las listas, la información es encerrada entre corchetes.
Dentro de los corchetes van la información, separada por comas, por lo que a esta información denominamos “elementos”. Es decir, en este ejemplo, tenemos 3 elementos, y todos son de distinto tipo de dato. El primer elemento es un *string*, el segundo *integer* y el tercero *bool*.

Estructuras de datos: Lista → características

Características de las listas

Como se ve en el ejemplo anterior, las listas pueden almacenar distintos tipos de datos, siempre separados por coma y cada uno se llama elemento.
Esto quiere decir que las listas están indexadas, por lo que si uno quiere acceder a un solo elemento de la lista, puede acceder mediante su índice:

- **Las listas son ordenadas e indexadas**: Esto quiere decir que una lista tiene un orden definido, por lo que se puede acceder a cualquier elemento de la lista mediante un índice numérico, comenzando desde el cero.
  La indexación funciona como una fila de casilleros numerados, donde cada casillero, contiene un elemento de la lista.



Por lo que, si se quiere acceder a un elemento de la lista, se debe llamar a la lista e incluir el número del elemento a que se quiere llamar:

```
lista.py > ...
1   lista = ['dato', 123, True]
2
3   print(lista[0])
4   print(lista[1])
5   print(lista[2])
```

```
PS C:\Us
thon.exe
dato
123
True
PS C:\Us
```

Estructuras de datos: Lista → características

Características de las listas

- **Las listas son mutables**: Esta característica nos permite modificar elementos de una lista después de haberla creado. Tal cual se comportan las variables, que, durante la ejecución del programa pueden modificar su valor, lo mismo pasa con los elementos de una lista.

```
1. lista = ['dato', 123, True]
2. lista = [23.8, False, 'Hola']
3. lista = [2030, 'Hola Info', 'valor']
```

- **Las listas permiten datos duplicados**: A diferencia de otras estructuras de datos que veremos un poco más adelante, las listas si permiten tener elementos duplicados.

```
lista = ['dato', 'dato', 'dato']
```

- **Las listas son dinámicas**: Las listas pueden contener diferentes tipos de datos, como lo vimos hasta ahora, pero además, pueden soportar otros objetos. Esto significa que pueden soportar paquetes multidimensionales de datos, como un otras listas u otras estructuras de control.
  Veamos un ejemplo de esto en Python.

Estructuras de datos: Lista → características

lista.py > ...

```python
lista_1 = ['dato', 123, True]

lista_2 = ['Diego', 'Ceci', 'Franco']

lista_3 = ['Un dato', 'Una lista', lista_1, 'Otra lista', lista_2]

print(lista_1, '\n')
print(lista_2, '\n')
print(lista_3, '\n')
```

PROBLEMAS SALIDA CONSOLA DE DEPURACIÓN TERMINAL PUERTOS

```
PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/AppData/Local/Programs/Python/Python31
['dato', 123, True]

['Diego', 'Ceci', 'Franco']

['Un dato', 'Una lista', ['dato', 123, True], 'Otra lista', ['Diego', 'Ceci', 'Franco']]
```

Vamos a analizar todo lo que está ocurriendo aquí, en la próxima página.

Estructuras de datos: Lista → características

```python
lista_1 = ['dato', 123, True]

lista_2 = ['Diego', 'Ceci', 'Franco']

lista_3 = ['Un dato', 'Una lista', lista_1, 'Otra lista', lista_2]
```

› Primero declaramos 3 listas con diferentes tipos de datos (lista_1, lista_2, lista_3).

› En “lista_3” como se puede observar, además de agregar elementos de tipo *string,* también estamos agregando las listas declaradas anteriormente.

```python
print(lista_1, '\n')
print(lista_2, '\n')
print(lista_3, '\n')
```

› Luego usamos la función print() para mostrar los valores que tienen las listas.

Lo que ves en con “\n” (barra invertida + n) son caracteres especiales que se usan para dar formato al texto, no se imprimen por pantalla. En este caso “\n” sirve para hacer saltos de línea, y “\t” no serviría para hacer una sangría o tabulación.

```
['dato', 123, True]

['Diego', 'Ceci', 'Franco']

['Un dato', 'Una lista', ['dato', 123, True], 'Otra lista', ['Diego', 'Ceci', 'Franco']]
```

› Cuando ejecutamos el programa, vemos en consola como se muestran los datos. En las dos primeras líneas de impresión vemos cada elemento de la lista, encerrado entre paréntesis, separado por coma y si es *string* entre comillas. Pero en la tercer línea de impresión, donde estamos mostrando la *lista_3,* observamos los elementos de la misma forma que las impresiones anteriores pero en donde están las listas (*lista_1* y *lista_2*) vemos nuevos corchetes, encerrando los elementos de las listas que se encuentran dentro de la *lista_3.*

De esta forma, podemos decir que, si queremos acceder a los elementos de las listas, dentro de la lista, ya no sería posible de manera individual, ya que la lista misma se convierte en un solo elemento.

Estructuras de datos: Lista → características

›Si accedemos a la *lista_1* o a la *lista_2* mediante su índice, vemos lo explicado anteriormente.

```
7   print(lista_3[2])
8   print(lista_3[4])

PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   T

PS C:\Users\Pc\Documents\Info\Python> &
['dato', 123, True]
['Diego', 'Ceci', 'Franco']
PS C:\Users\Pc\Documents\Info\Python>
```

Una lista también se puede recorrer mediante ***slicing***, que es una técnica que se utiliza para obtener subconjuntos de elementos de la lista.

Supongamos que solo queremos ver los elementos de las posiciones 2, 3 y 4, entonces utilizarémos el *slicing*.

```
24  colores = ['azul', 'blanco', 'rosa', 'verde', 'negro', 'lila', 'morado']
25  print(colores[2:5])
26
27

PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL   PUERTOS

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/AppData/Local/Programs/Python
['rosa', 'verde', 'negro']
```

Como el elemento de la última posición que marcamos como índice no es devuelto, deberemos poner siempre un número más. En este ejemplo queríamos los elementos de las posiciones 2, 3 y 4, por lo que nuestro argumento en el slicing debe ser: [2:**5**].

Para ver todas las formas de uso de slicing, te dejamos una documentación de referencia con varios ejemplos:
https://python-reference.readthedocs.io/en/latest/docs/brackets/slicing.html

Estructuras de datos: Lista → características

Las listas también se pueden recorrer en reversa, es decir, desde el último elemento hacía el primer elemento. Esto se puede hacer poniendo el número de índice pero de forma negativa.

```
24  colores = ['azul', 'blanco', 'rosa', 'verde']
25  print(colores[-1])
26  print(colores[-2])
27
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUE

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/A
verde
rosa
```

Además, usando un índice, podemos agregar un elemento a una lista.

```
24  colores = ['azul', 'blanco', 'rosa', 'verde']
25  print(colores, '\n')
26  colores[2] = 'negro'
27  print(colores)
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUE

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/A
['azul', 'blanco', 'rosa', 'verde']

['azul', 'blanco', 'negro', 'verde']
```

Y lo mismo se puede hacer usando un índice negativo.

```
24  colores = ['azul', 'blanco', 'rosa', 'verde']
25  print(colores, '\n')
26  colores[-1] = 'negro'
27  print(colores)
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUE

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/
['azul', 'blanco', 'rosa', 'verde']

['azul', 'blanco', 'rosa', 'negro']
```

Estructuras de datos: Lista → características

También podemos agregar otras listas o incluso otras estructuras de datos (que veremos más adelante).

```
24  colores = ['azul', 'blanco', 'rosa', 'verde']
25  frutas = ['naranja', 'manzana', 'banana']
26  colores[2] = frutas
27  print(colores)
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/AppData/L
['azul', 'blanco', ['naranja', 'manzana', 'banana'], 'verde']
```

```
24  colores = ['azul', 'blanco', 'rosa', 'verde']
25  frutas = ['naranja', 'manzana', 'banana']
26  colores[-1] = frutas
27  print(colores)
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/AppData
['azul', 'blanco', 'rosa', ['naranja', 'manzana', 'banana']]
```

Métodos de una lista

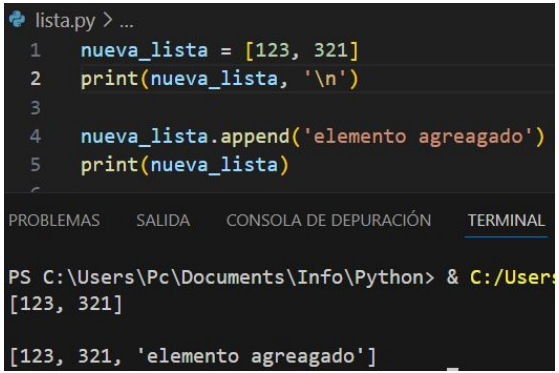
Las estructuras de datos, en este caso, una lista tienen muchos métodos, sin embargo, vamos a ver algunos de los más comunes que existen.

▶ Algo importante a aclarar es que, un método no es lo mismo que una función ya que, una función puede usarse de manera global, pero un método sólo puede llamarse desde un objeto específico (estos conceptos lo verás a profundidad en el material complementario). ▶

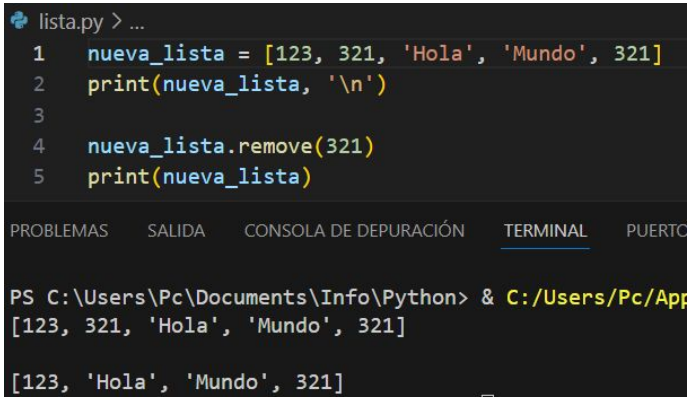
Uno de los métodos que más vamos a usar sobre una lista es el de agregar elementos dentro de ella, como en el siguiente ejemplo:

Estructuras de datos: Lista → métodos

- Método **append(*parámetro*)** : Este método sirve para agregar un elemento al final de la lista, tal cual vemos en el ejemplo más abajo.
  Todos los métodos irán con la siguiente nomenclatura que es, escribir el nombre de la lista y un punto, el nombre del método y, generalmente, un parámetro dentro de los paréntesis.



- Método **remove(*parámetro*)** : Mediante este método, se buscará y se eliminará, dentro de la lista, el primer elemento que se encuentre con el valor del parámetro que le asignemos.



En este ejemplo se puede ver que el valor “321” aparece dos veces, pero solo es eliminado el que se encuentra en la posición 1 (*nueva_lista[1]*).

Estructuras de datos: Lista → métodos

Como te lo mencionamos antes, las listas cuentan con varios métodos, pero te dejamos la lista de otros métodos comunes para que los puedas probar, y en esta clase podamos seguir viendo otras estructuras de datos.

- Otro métodos comunes de listas:
  - **clear()** : Elimina todos los elementos de la lista.
  - **copy()** : Devuelve una copia superficial de la lista.
  - **count(*parámetro*)** : Devuelve el número de elementos con el valor del argumento pasado.
  - **extend(*iterable*)** : Extiende la lista agregando todos los elementos de un iterable al final de la lista (puede ser otra lista u otra estructura de datos).
  - **index(*parámetro*)** : Devuelve el número de índice en el que se encuentra el argumento.
  - **insert(*posición*, *parámetro*)** : Inserta un elemento en una posición determinada. *Posición* llevará el número de la posición del índice donde se quiere insertar el *argumento*.
  - **pop(*posición* = -1)** : Elimina y devuelve el elemento de una posición determinada. El valor predeterminado es el último elemento.
  - **reverse()** : Invierte el orden de los elementos de la lista en su lugar.
  - **sort(*key = None, reverse = False*)** : Ordena los elementos de la lista en su lugar (orden ascendente predeterminado).

¡Te invitamos a probar estos métodos en clases de mentorías! Además, vas a ver que varios de estos métodos pueden aplicarse a otras estructuras de datos que veremos a continuación.

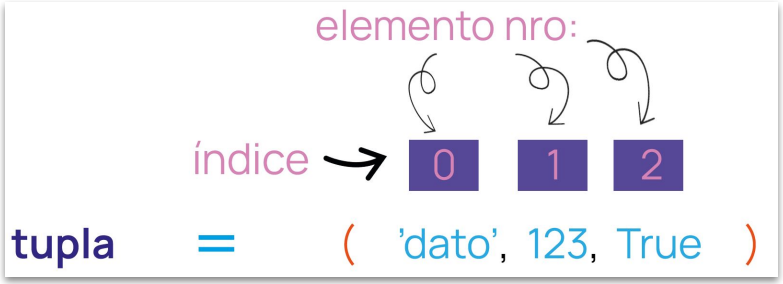
Estructuras de datos: Tupla - tuple

Tuplas en Python

Estas estructuras son similares a las listas, pero con una gran diferencia: las tuplas no se pueden modificar una vez creadas.

Igual que las listas, las tuplas son estructuras de datos que permiten almacenar una colección de elementos **ordenados**, por lo que se pueden acceder a ellos mediante índices, pero como te mencionámos antes, estas son **inmutables**.

- **Las tuplas son inmutables**: Esto significa que una vez que se crea una tupla, no se pueden modificar, los valores de sus elementos, ni tampoco, agregar o eliminar.



Otra diferencia visible en la estructura básica de su sintaxis es que, los elementos de una tupla se encierran dentro de paréntesis, a diferencia de una lista que llevaba corchetes. En realidad, a una tupla puede crearse sin paréntesis, solo asignando elementos y separándolos por comas, pero por convención usamos paréntesis.

Lo que brinda una tupla, a diferencia de una lista es:

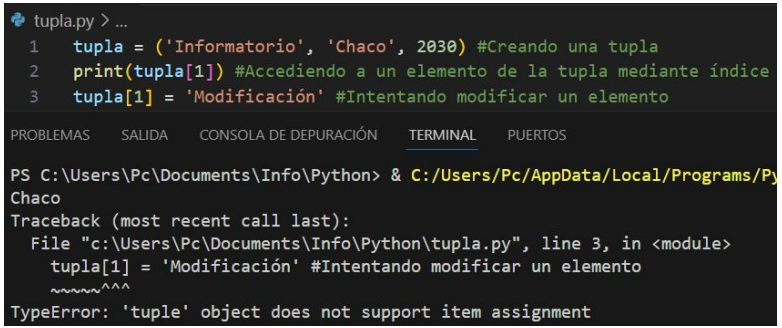
Estructuras de datos: Tupla → características

- **Seguridad**: Las tuplas son inmutables, lo que las hace más seguras que las listas para almacenar datos que no deben ser modificados accidentalmente. Esto es particularmente útil en situaciones donde la integridad de los datos es crítica.

- **Eficiencia**: Las tuplas son más eficientes en memoria que las listas, ya que Python no necesita almacenar información adicional sobre la posibilidad de modificación.

- **Claridad**: El uso de tuplas indica claramente que los datos almacenados no deben ser cambiados, mejorando la legibilidad y mantenibilidad del código.

Ejemplo de uso de una Tupla



Como podés ver en el ejemplo, creamos una tupla en la primer línea de código, la mostramos por pantalla usando un índice en la segunda línea de código, e intentamos modificar un elemento, mediante su índice, en la tercer línea de código, pero esto nos arroja un error en consola, ya que una tupla, una vez creada, no se puede modificar.

Estructuras de datos: Tupla → características

Para darte una idea, las tuplas son realmente útiles cuando por ejemplo, necesitamos almacenar un registro de nombres, apellidos, etc, es decir, datos sensibles que no deben ser eliminados.

La mayoría de métodos usados en una lista, pueden ser usados en una tupla, excepto, los que intentan modificar o eliminar elementos de la tupla.

Algo a tener en cuenta es que si bien, no se pueden modificar elementos de la tupla una vez creada, sí se puede cargar elementos nuevos a una tupla una vez creada, pero esto significa que los elementos anteriores serán eliminados. Te lo mostramos en el siguiente ejemplo:

```
tupla.py > ...
1   tupla = ('Informatorio', 'Chaco', 2030) #Creando una tupla
2   print(tupla, '\n') #Mostrando por consola
3   tupla = ('Cargando', 'nuevos', 'datos') #Reemplazando los elementos
4   print(tupla) #Mostrando por consola

PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/AppData/Local/Programs
('Informatorio', 'Chaco', 2030)

('Cargando', 'nuevos', 'datos')
```

Entonces debemos tener en cuenta que, si bien las tuplas son inmutables, se pueden reemplazar todos los elementos.

En clases de mentorías, te invitamos a aplicar, a las tuplas, los métodos que viste para listas.
Además, podés intentar otras cosas que viste con variables, como concatenar 😉

Estructuras de datos: Conjunto - set

Conjuntos en Python

Esta estructura de datos, a diferencia de las listas y tuplas, almacena elementos pero de manera **desordenada** y sus elementos son **únicos**.
En las listas y tuplas cada elemento estaba ordenado, es decir, podíamos acceder a ellos mediante un índice, y además, se podían duplicar, es decir, los mismos valores de los elementos podían aparecer más de una vez, pero en los conjuntos, no se permiten duplicados, cada valor de un elemento es único.
Para que te dés una idea de cómo funcionan los conjuntos, imaginate que tenés una canasta de frutas, pero esta canasta tiene la particularidad de que, por más veces que pongas una naranja adentro, siempre va a haber una sola naranja.

Para declarar un conjunto, seguimos con la misma estructura básica, pero en vez de usar corchetes o paréntesis, se usan llaves de apertura y cierre.

Los conjuntos utilizan la estructura de datos llamada tabla de hash, que permite almacenar elementos de manera eficiente para una rápida búsqueda y verificación de unicidad.
Ya que los conjuntos no tienen un orden definido, no se puede acceder a un elemento de la misma manera que lo hacíamos en las listas o tuplas.
Para el caso de conjuntos, vamos a usar estructuras de control como condicionales y bucles. De esta forma, podemos recorrer un conjunto en busca de un elemento:

Estructuras de datos: Conjunto → recorriendo esta estructura de datos

```
conjunto.py > ...
1   conjunto = {'Info', 'Chaco', 'Info', 123, 123}
2   print(conjunto)

PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERT

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/Ap
{123, 'Chaco', 'Info'}
```

Primero que nada, creamos el conjunto y lo mostramos por consola. Vemos en la salida que el orden de los elementos no es el mismo que cuando asignamos esos datos.
Además, apreciamos que los datos que eran duplicados, ya no están ya que esta estructura se encargará de eliminar los datos duplicados.
Para recorrer un conjunto en busca de un elemento específico, podemos hacerlo de dos maneras. En el siguiente ejemplo vamos a usar un condicional.

```
conjunto.py > ...
1   conjunto = {'Info', 'Chaco', 'Info', 123, 123}
2   print(conjunto)
3
4   if 'Info' in conjunto:
5       print('El valor "Info" se encuentra en el conjunto.')
6   else:
7       print('El valor "Info" NO se encuentra en el conjunto.')

PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/AppData/Local/Pr
{123, 'Info', 'Chaco'}
El valor "Info" se encuentra en el conjunto.
```

Para este ejemplo usamos un condicional recorriendo el conjunto mediante “in” (si, “in” podemos usarlo con *if* o con *for*), ya que el operador “in” es un operador de pertenencia que se utiliza para verificar si un elemento específico está presente dentro de una secuencia de datos (puede ser una lista, tupla,conjunto, incluso una variable).
Para este ejemplo, el condicional funcionó por el bloque de código *verdadero* ya que el valor se encontró dentro del conjunto, pero si no lo encontraba, hubiese ido por el bloque de *falso*.

Estructuras de datos: Conjunto → recorriendo esta estructura de datos

```python
conjunto = {'Info', 'Chaco', 'Info', 123, 123}
print(conjunto)

valor_buscado = 'Chaco'

if valor_buscado in conjunto:
    print('El valor', valor_buscado,'se encuentra en el conjunto.')
else:
    print('El valor', valor_buscado,'NO se encuentra en el conjunto.')
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/AppData/Local/Programs/Pyt
{'Info', 123, 'Chaco'}
El valor Chaco se encuentra en el conjunto.
```

Otra forma de buscar un elemento sería usando una variable, como en el ejemplo que acabamos de ver, pero para recorrer cada elemento de un conjunto usamos *for*.

```python
conjunto = {'Info', 'Chaco', 'Info', 123, 123}

for i in conjunto:
    print(i)
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/App
Chaco
123
Info
```

Vemos que usando *for* hacemos el recorrido de cada elemento, y, para este caso hacemos la impresión de los elementos en cada iteración.

Otra cosa que se aprecia es que, la impresión de los elementos no sale en el mismo orden que fueron almacenados, ya que, repetimos, no es una estructura de datos ordenada.

Estructuras de datos: Conjunto → operaciones con conjuntos

Hay operaciones que son realmente útiles cuando usamos conjuntos, por lo que veremos algunas.

- **Unión (|)** : Mediante esta operación, podemos combinar dos conjuntos en un nuevo (similar a concatenar).

```
conjunto.py > ...
1  frutas = {'naranja', 'manzana', 'pera'}
2  verduras = {'lechuga', 'tomate', 'zanahoria'}
3
4  frutas_y_verduras = frutas | verduras
5
6  print(frutas_y_verduras)

PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/AppData/Local/
{'manzana', 'pera', 'zanahoria', 'tomate', 'naranja', 'lechuga'}
```

- **Intersección (&)** : Esta operación es ideal cuando necesitamos crear un nuevo conjunto con elementos que están presentes en dos o más conjuntos.

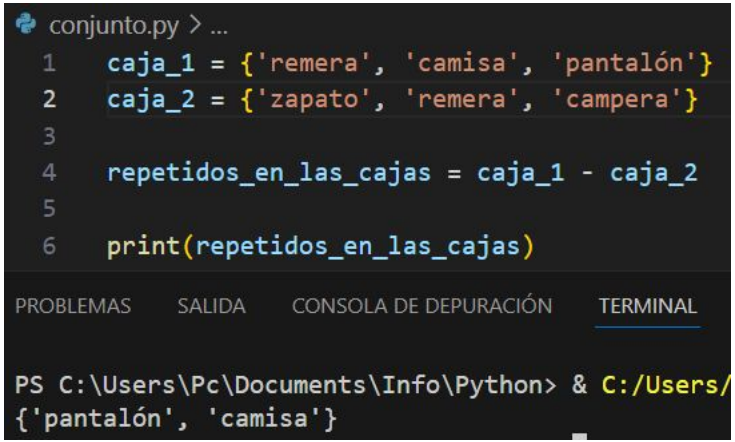
```
conjunto.py > ...
1  caja_1 = {'remera', 'camisa', 'pantalón'}
2  caja_2 = {'zapato', 'remera', 'campera'}
3
4  repetidos_en_las_cajas = caja_1 & caja_2
5
6  print(repetidos_en_las_cajas)

PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/
{'remera'}
```

Estructuras de datos: Conjunto → operaciones con conjuntos

- **Diferencia (-)** : A diferencia de la operación de intersección, con esta operación, podemos crear un nuevo conjunto con los elementos que se encuentran en el primer conjunto, pero no en el segundo.



>>Además te dejamos una listas de funciones útiles para los conjuntos:

- **len(conjunto)** : Devuelve la cantidad de elementos en el conjunto. Esta función se puede usar en listas, tuplas e incluso variables.
- **conjunto.add(elemento)** : Agrega un elemento al conjunto si no está presente.
- **conjunto.remove(elemento)** : Elimina un elemento del conjunto si está presente.
- **conjunto.discard(elemento)** : Intenta eliminar un elemento del conjunto (sin error si no está presente).
- **conjunto.clear()** : Vacía el conjunto, eliminando todos sus elementos.
- **conjunto.copy()** : Crea una copia del conjunto original.

Estructuras de datos: Diccionario - dict

Diccionarios en Python

Por último, tenemos esta estructura de datos capaz de almacenar colecciones de **pares clave - valor**.
En esta estructura, los elementos son identificados mediante una clave con valor único, por lo que de esta manera permite realizar una búsqueda rápida y eficiente por clave. Además, sus valores son mutables.

Para darte una idea de su funcionamiento, es similar a lo que hacés cada vez que buscás el número de teléfono de un contacto en tu celular, lo buscás poniendo el nombre de como lo tenés agendado (esa sería tu clave).
Los diccionarios también utilizan la tabla de hash, por lo que son muy eficientes.

Para ingresar sus pares clave - valor, éstos van a estar encerrados entre llaves (como sucedía con los conjuntos).
Además, para separar, lo que es la clave del valor, se deben usar dos puntos (:), tal cual vemos en el ejemplo de muestra.

Veamos un ejemplo en Python de cómo creamos un diccionario.

Estructuras de datos: Diccionario - dict

```python
diccionario = {'país' : 'Argentina',
               'provincia' : 'Chaco',
               'ciudad' : 'Resistencia',
               'cp': 3500}

print(diccionario)
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/AppData/Local/Programs/Python
{'país': 'Argentina', 'provincia': 'Chaco', 'ciudad': 'Resistencia', 'cp': 3500}
```

También podemos acceder a un valor, usando su clave:

```python
diccionario = {'país' : 'Argentina',
               'provincia' : 'Chaco',
               'ciudad' : 'Resistencia',
               'cp': 3500}

print(diccionario['país'])
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUE

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/
Argentina
```

O si queremos agregar un par clave valor a un diccionario, lo podemos hacer de la siguiente manera:

```python
diccionario = {'país' : 'Argentina',
               'provincia' : 'Chaco',
               'ciudad' : 'Resistencia',
               'cp': 3500}

print(diccionario)

diccionario['departamento'] = 'San Fernando'
print(diccionario)
```

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/AppData/Local/Programs/Python/Python312/python.exe c:/Users/P
{'país': 'Argentina', 'provincia': 'Chaco', 'ciudad': 'Resistencia', 'cp': 3500}
{'país': 'Argentina', 'provincia': 'Chaco', 'ciudad': 'Resistencia', 'cp': 3500, 'departamento': 'San Fernando'}
```

O si necesitamos modificar un valor lo hacemos llamando a su clave y asignando el nuevo valor:

Estructuras de datos: Diccionario - dict

```
diccionario.py > ...
1   curso = {'nombre' : 'Info',
2            'etapa' : 1}
3
4   print(curso)
5   curso['etapa'] = 2
6   print(curso)

PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN

PS C:\Users\Pc\Documents\Info\Python> &
{'nombre': 'Info', 'etapa': 1}
{'nombre': 'Info', 'etapa': 2}
```

Algo que también es muy similar a los conjuntos, es la forma en la que podemos recorrer los diccionarios, ya que lo podemos hacer utilizando condicionales o bucles.

Por ejemplo, podemos verificar si una clave se encuentra en un diccionario, usando condicional:

```
diccionario.py > ...
1   curso = {'nombre' : 'Info',
2            'etapa' : 2}
3
4   if 'nombre' in curso:
5       print('La clave "nombre" se encuentra en el diccionario.')
6   else:
7       print('La clave "nombre" NO se encuentra en el diccionario.')

PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL   PUERTOS

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/AppData/Local/Progra
La clave "nombre" se encuentra en el diccionario.
```

O podemos recorrer los pares clave - valor usando el bucle *for:*

Estructuras de datos: Diccionario - dict

```
diccionario.py > ...
1   curso = {'nombre' : 'Info',
2            'etapa' : 2}
3
4   for clave, valor in curso.items():
5       print(f'Clave: {clave}. Valor: {valor}')

PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TERMINAL   PU

PS C:\Users\Pc\Documents\Info\Python> & C:/Users/Pc/
Clave: nombre. Valor: Info
Clave: etapa. Valor: 2
```

Hay dos cosas para resaltar en este ejemplo. Lo primero es que, como el bucle *for* comprende que estamos recorriendo un diccionario, podemos usar dos variables para las iteraciones del bucle. Por otro lado, estamos usando la función ".items()" que nos sirve para ver los *items* que conforman al diccionario.

Lo otro para resaltar y que, lo vamos a usar de ahora en adelante, es una forma de mostrar resultados usando "f-string".
Como vés, antes de poner las comillas para escribir texto que se mostrará por consola con *print()*, vamos a agregar la letra "f" minúscula, y cuando vamos a mostrar los valores que necesitamos mostrar, pondremos esas variables encerradas en llaves ({clave}, {valor}). Esto lo vamos a usar en lugar de usar una coma y luego la variable a mostrar:

```
1   variable = 'Informatorio'
2   print(f'Bienvenido al {variable}.')

PROBLEMAS   SALIDA   CONSOLA DE DEPURACIÓN   TE

PS C:\Users\Pc\Documents\Info\Python> & C:
Bienvenido al Informatorio.
```

Estructuras de datos: Diccionario - dict

Por último, y siguiendo con diccionarios, si queremos obtener las claves o valores, lo haremos de la siguiente forma:

Usando las funciones sobre el diccionario ".keys()" y ".values()", obtendremos las claves y valores respectivamente. Estos datos los guardamos en variables, pero además, usamos otra función más, que es la de convertir la variable en una lista mediante la palabra reservada "list".

Así como hacíamos conversiones con los tipos de datos básicos (str, int, float, bool), también lo podemos realizar con las estructuras de datos: Para eso usamos las palabras reservadas:

- **list**: De esta forma podemos convertir en **lista**.
- **tuple**: De esta forma podemos convertir en **tupla**.
- **set**: De esta forma podemos convertir en **conjunto**.
- **dict**: De esta forma podemos convertir en **diccionario**.

Hay ciertas formas a seguir para hacer las conversiones, pero todo lo que no pudimos mostrar en esta clase, los mentores podrán mostrarlo en clase de mentoría.

Lo que vimos hoy 😉

En esta clase vimos las estructuras de datos fundamentales, su uso, algunas funciones y métodos que pueden usarse sobre estas estructuras.

Hay varios métodos de los cuáles vimos sobre listas, que pueden usarse en otras estructuras, por lo que te invitamos a asistir a mentorías para que tu mentor pueda estar realizando algunos ejemplos y comprendas mejor el uso de estas estructuras, sus métodos y funciones.

Además, no olvides que en el material complementario podrás encontrar más ejemplos y profundizar en todos los contenidos.

¡Nos vemos En la próxima clase!