CONJUNTOS "Set".

Un conjunto o **set** es una **colección no ordenada de objetos únicos**, es decir, no tiene elementos duplicados. Python provee este tipo de datos por defecto al igual que otras colecciones más convencionales como las **listas**, **tuplas** y **diccionarios**.

El conjunto se describe como una lista de ítems separados por coma y contenido entre dos llaves.

```
>>> conjunto =  {1, 2, 3, 4}
>>> otro_conjunto = {"Hola", "como", "estas", "?"}
>>> conjunto_vacio = set()   #{ } [ (  ] )
```

Para crear un conjunto vacío debemos decirle **set()** de lo contrario si quisiéramos hacer como las listas y crearlo con {} python crea un diccionario, el cual veremos más adelante

Heterogéneos.

En otros lenguajes, las colecciones tienen una restricción la cual sólo permite tener un sólo tipo de dato. Pero en Python, no tenemos esa restricción. Podemos tener un **conjunto heterogéneo** que contenga números, variables, strings, o tuplas.

Ejemplo:

```
>>> mi_var = 'Una variable'
>>> datos = {1, -5, 123.1,34.32, 'Una cadena', 'Otra cadena', mi_var}
```

Sin embargo, un conjunto **no** puede incluir objetos mutables como listas, diccionarios, e incluso otros conjuntos o **set**.

Ejemplo:

```
>>> s = {{1,2}, [1,2,3,4],2}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'
```

De la misma forma podemos obtener un conjunto a partir de cualquier objeto **iterable**:

```
>>> set1 = set([1, 2, 3, 4])
{1, 2, 3, 4}
>>> set2 = set(range(10))
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Set

Un set puede ser **convertido** a una **lista** y **viceversa**. En este **último** caso, los elementos **duplicados** son **unificados**.

```
>>> list({1, 2, 3, 4})
[1, 2, 3, 4]
>>> set([1, 1, 2, 2, 3, 3, 4, 4])
{1, 2, 3, 4}
```

List vs. Set

Como hablamos, las listas son **mutables**, sin embargo, el set también es **mutable**, pero no podemos hacer slicing, ni manejar un set por índice.

Ejemplo:

```
>>> conjunto = {‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’}
>>> conjunto[:3] = [‘A’, ‘B’, ‘C’]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not subscriptable
```

FUNCIONES DE CONJUNTOS

Funciones Integradas

En los conjuntos, hay funciones que son muy interesantes e importantes, las **funciones integradas**.

Los conjuntos en python tienen muchas funciones para utilizar, entre todas ellas vamos a nombrar las más importantes.

Add

La primer función de los conjuntos de la que estamos hablando es **ADD**. Esta función permite agregar un nuevo ítem al set. La misma se escribe mi_conjunto**.add(**ítem_a_agregar**)**

mi_conjunto sería el set al que se le desee agregar el ítem, e ítem_a_agregar sería el ítem que deseemos agregar al set.

Ejemplo:

```
>>> numeros =  {1,2,3,4}
>>> numeros.add(5)
{1,2,3,4,5}
```

No sólo acaba ahí. En la función add también podemos realizar operaciones aritméticas en nuestro ítem.

Ejemplo:

```
>>> numeros =  {1,2,3,4}
>>> numeros.add(3*2)
{1,2,3,4,6}
>>> numeros.add(3**2+1-12+5*)
```

{1,2,3,4,6,13}

Update

Para añadir múltiples elementos a un set se usa la función **update**(), que puede tomar como argumento una lista, tupla, string, conjunto o cualquier objeto de tipo iterable.
La misma se escribe: mi_conjunto**.update(**ítem_a_agregar**)**

```
>>> numeros =  {1,2,3,4}
>>> numeros.update([5,6,7,8])
{1,2,3,4,5,6,7,8}
>>> numeros.update(range(9,12))
{1,2,3,4,5,6,7,8,9,10,11}
```

Longitud del set

¿Se acuerdan cuando hablamos de **len** en listas?

En set, se puede usar exactamente la misma función para poder saber la longitud de un set, es decir, la cantidad de ítems dentro del mismo.

Ejemplo:

```
>>> numeros =  {1,2,3,4}
>>> len(numeros)
4
>>> datos = {1, -5, 123.34, 'Una cadena', 'Otra cadena'}
>>> len(datos)
5
```

Discard

Si **add** te deja agregar un ítem al set, **discard** hace todo lo contrario, elimina el ítem del set, sin modificar el resto del set, si el elemento pasado como argumento a **discard**() no está dentro del conjunto es simplemente ignorado.
Se escribe como mi_conjunto**.discard(**item_a_descartar**)**.

```
>>> numeros =  {1, 2, 3, 4}
>>> numeros.discard(2)
{1, 3, 4}
>>> datos = {1, -5, 123,34, 'Una cadena', 'Otra cadena'}
>>> datos.discard('Otra cadena')
{1, -5, 123,34, 'Una cadena'}
```

Remove

La función **remove** funciona igual al discard, pero con una diferencia, en discard si el ítem a remover no existe, simplemente se ignora. En **remove** en este caso nos indica un error.

Se escribe como mi_conjunto**.remove(**item _a_ remove**)**.

```
>>> numeros =  {1, 2, 3, 4}
>>> numeros.remove(2)
```

```
{1, 3, 4}
>>> numeros.remove(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 5
```

In

Para determinar si un elemento **pertenece** a un set, utilizamos la palabra reservada **in**.
Se escribe como item_a_validar **in** mi_conjunto

```
>>> numeros =  {1, 2, 3, 4}
>>> 2 in numeros
True
>>> 2 not in numeros
False
>>> 4 in numeros
False
```

Clear

Igual que en las listas, podremos borrar todos los valores de un set simplemente usando la función **clear**.
Se escribe como mi_conjunto.**clear**().

¡No se puede asignar un set vacío por que lo toma como diccionario!

```
>>> numeros =  {1, 2, 3, 4}
>>> numeros.clear()
set()
```

Pop

La función **pop** retorna un elemento en forma aleatoria (no podría ser de otra manera ya que los elementos no están ordenados). Así, el siguiente bucle imprime y remueve uno por uno los miembros de un conjunto.

```
>>> numeros =  {1,2,3,4}
>>> while numeros:
        print("Se está borrando: ", numeros.pop())
Se está borrando:  1
Se está borrando:  2
Se está borrando:  3
Se está borrando:  4
```

DICCIONARIOS

Un diccionario **dict** es una colección no ordenada de objetos. Es por eso que para identificar un valor cualquiera dentro de él, especificamos una **clave** (a diferencia de las listas y tuplas, cuyos elementos se identifican por su posición).

Las **claves** suelen ser **int** o **string**, aunque cualquier otro objeto inmutable puede actuar como una clave. Los valores, por el contrario, pueden ser de cualquier tipo, incluso otros diccionarios.

Para crear un diccionario se emplean llaves {}, y sus pares clave-valor se separan por comas. A su vez, intercalamos la clave del valor con dos puntos (:)
Nota: Para crear un diccionario vacío se puede hacer diccionario = {}

```
>>> colores = {"amarillo": “yellow”, "azul": “blue”, "rojo": “red”}
{"amarillo": “yellow”, "azul": “blue”, "rojo": “red”}
>>> type(colores)
<class 'dict'>
```

Para traer el valor de un **diccionario** se utiliza su **clave**

```
>>> colores = {"amarillo": “yellow”, "azul": “blue”, "rojo": “red”}
>>> colores[“amarillo”]
“yellow”
>>> colores[“azul”]
“blue”
>>> numeros = {10:”diez”, 20:”veinte”}
>>> numeros[10]
“diez”
```

Mutabilidad

Los diccionarios al igual que las listas son **mutables**, es decir, que podemos reasignar sus ítems haciendo referencia con el índice.

```
>>> colores = {"amarillo": “yellow”, "azul": “blue”, "rojo": “red”}
>>> colores[“amarillo”] = “white”
>>> colores[“amarillo”]
“white”
```

Asignación

También permite operaciones en asignación

```
>>> edades = {"Juan": 26, "Esteban": 35, "Maria": 29}
>>> edades[“Juan”] += 5
>>> edades[“Juan”]
31
>>> edades[“Maria”] *= 2
>>> edades[“Maria”]
58
```

Funciones de Diccionarios

Al igual que en conjuntos, en los diccionarios encontramos **funciones integradas**.

Los diccionarios en python tienen muchas funciones para utilizar. Si bien hablaremos las desarrollaremos más adelante, a continuación vamos a nombrar las más importantes.

Add

No hay una función de add, pero para agregar una nueva clave-valor se puede realizar de la siguiente manera:

```
>>> numeros =  {“uno”: 1, ”dos”: 2, “tres”: 3, “cuatro”: 4}
>>> numeros[“cinco”] = 5
{“uno”: 1, ”dos”: 2, “tres”: 3, “cuatro”: 4, “cinco”: 5}
```

En este caso, creamos una nueva clave que no existe “**cinco**” y asignamos el valor **5**

Update

Este método actualiza un diccionario agregando los pares clave-valores.

```
>>> numeros =  {“uno”: 1, ”dos”: 2, “tres”: 3, “cuatro”: 4}
>>> numeros.update({“cinco”: 5, ”seis”: 6})
{“uno”: 1, ”dos”: 2, “tres”: 3, “cuatro”: 4, “cinco”: 5, ”seis”: 6}
>>> otro_dict = dict(siete=7)
>>> numeros.update(otro_dict)
{“uno”: 1, ”dos”: 2, “tres”: 3, “cuatro”: 4, “cinco”: 5, ”seis”: 6, “siete”: 7}
```

El método update() **toma un diccionario o un objeto iterable de pares clave/valor (generalmente tuplas).** Si se llama a update() sin pasar parámetros, el diccionario permanece sin cambios.

Longitud del diccionario

¿Se acuerdan cuando hablamos de **len** en listas? En dict, se puede usar exactamente la misma función para poder saber la longitud de un dict, es decir, la cantidad de ítems dentro del mismo.

```
>>> numeros =  {“uno”: 1, ”dos”: 2, “tres”: 3, “cuatro”: 4}
>>> len(numeros)
4
```

Del

Del elimina el ítem del dict, sin modificar el resto del dict, si el elemento pasado como argumento a **del**() no está dentro del dict es simplemente ignorado.

Se escribe como **del** mi_dict[“**clave**”].

```
>>> numeros =  {“uno”: 1, ”dos”: 2, “tres”: 3, “cuatro”: 4}
>>> del numeros[“dos”]
{“uno”: 1, “tres”: 3, “cuatro”: 4}
```

In

Para determinar si un elemento **pertenece** a un dict, utilizamos la palabra reservada **in**.
Se escribe como clave_a_validar **in** mi_dict

```
>>> numeros =  {“uno”: 1, ”dos”: 2, “tres”: 3, “cuatro”: 4}
>>> “dos” in numeros
True
>>> 2 not in numeros
False
>>> 4 in numeros
False
```

Clear

Igual que en las listas, podremos borrar todos los valores de un dict simplemente usando la función **clear**.
Se escribe como dict.**clear**().

```
>>> numeros =  {“uno”: 1, ”dos”: 2, “tres”: 3, “cuatro”: 4}
>>> numeros.clear()
{}
```

Otra forma más cómoda es hacer mi_dict = {}

Pop

Este método remueve específicamente una clave de diccionario y devuelve valor correspondiente. Lanza una excepción KeyError si la clave no es encontrada.
Se escribe como mi_dict.**pop**(“**clave**”)

```
>>> numeros =  {“uno”: 1, ”dos”: 2, “tres”: 3, “cuatro”: 4}
>>> numeros.pop(“uno”)
{”dos”: 2, “tres”: 3, “cuatro”: 4}
```

Métodos de Colecciones

CADENAS “strings”

Upper

Esta función integrada sirve para hacer que se devuelva la misma cadena pero con sus caracteres en **mayúscula**, usando el método ***upper()***. Se escribe como: *string****.upper()***

```
>>> cadena = “Hola Mundo”
>>> cadena.upper()
“HOLA MUNDO”
>>> “hola amigo!”.upper()
“HOLA AMIGO!”
```

Lower

Ya vimos cómo convertir a mayúsculas, pero también es útil convertir una cadena de caracteres a **minúsculas**, usando el método ***lower()***. Se escribe como: *string****.lower()***

>>> cadena = “Hola Mundo”
>>> cadena.lower()
“hola mundo”
>>> “HoLa AmIgO!”.lower()
“hola amigo!”

Capitalize

Esta función integrada sirve para hacer que se devuelva la misma cadena pero con su **primer** carácter en **mayúscula** y el resto de caracteres hacerlos **minúscula**, usando el método ***capitalize****().* Se escribe como: *string****.capitalize()***

>>> cadena = “Hola Mundo”
>>> cadena.capitalize()
“Hola mundo”
>>> “HoLa AmIgO!”.capitalize()
“Hola amigo!”

Title

Esta función integrada sirve para hacer que se devuelva la misma cadena pero con el **primer** carácter de cada palabra en **mayúscula** y el resto de caracteres hacerlos **minúscula**, usando el método ***title****().* Se escribe como: *string****.title()***

>>> cadena = “hOLA mUNDO”
>>> cadena.title()
“Hola Mundo”
>>> “HoLa AmIgO!”.title()
“Hola Amigo!”

Count

Si necesitamos saber cuantas veces aparece una **subcadena** dentro de la misma **cadena**, usando el método ***count****().* Se escribe como: *string****.count()***

>>> cadena = “hOLa mUNDO esta cadena tiene muchas a”
>>> cadena.count(“a”)
6
>>> “HoLa amigo como estas amigo!”.count(“amigo”)
2

Find

Si necesitamos averiguar el índice en el que aparece una **subcadena** dentro de la misma **cadena**, usamos el método ***find****().* Se escribe como: *string****.find()****.* Si no encuentra la cadena devuelve un **-1**.

>>> cadena = “hOLa mUNDO esta cadena tiene muchas a”
>>> cadena.find(“esta”)
11
>>> “HoLa amigo como estas amigo!”.find(“chau”)
-1

Rfind

Es exactamente igual al método ***find()*** los diferencia en que ***rfind()*** devuelve el índice pero de la última ocurrencia de la subcadena, es decir, la última vez que aparece en la cadena. Se escribe como: *string****.rfind()***. Si no encuentra la cadena devuelve un **-1**.

>>> “HoLa amigo como estas amigo!”.find(“amigo”)
5
>>> “HoLa amigo como estas amigo!”.rfind(“amigo”)
22

Split

Esta función integrada sirve para devolver una lista con la cadena de caracteres separada por cada índice de la lista.
Se escribe como: *string****.split(****“cadena_a_separar”****)****.* Si no se indica alguna cadena para separar separa por “**espacios**”.

>>> cadena = “hOLA mUNDO”
>>> cadena.split()
[“hOLA”, “mUNDO”]
>>> “HoLa amigo como estas amigo!”.split(“amigo”)
[“HoLa”, “como ”, “estas ”, “!”]

Join

Esta función integrada sirve para devolver una cadena separada a partir de una especie de separador. Se escribe como: ***“separador”.join(****“cadena”****)****.*

Nota: Si no se especifica el separador nos devuelve un error
>>> cadena = “Hola mundo”
>>> “,”.join(cadena)
“H,o,l,a, ,m,u,n,d,o”
>>> “ ”.join(cadena)
“H o l a m u n d o”

Strip

Esta función integrada sirve para devolver una cadena borrando todos los caracteres delante y detrás de la cadena. Se escribe como: *cadena****.strip(****“caracter_a_borrar”****)****.*

Nota: Si no se especifica el carácter elimina los espacios
>>> cadena = “---------Hola mundo--------”
>>> cadena.strip(“-”)
“H,o,l,a, ,m,u,n,d,o”

>>> “ Hola mundo “.strip()
“Hola mundo”

Replace

Esta función integrada sirve para devolver una cadena reemplazando los sub caracteres indicados.
Se escribe como: *cadena**.replace(**“caracter_a_remplazar”, “caracter_que_reemplaza”**)**.*
También podemos indicar cuantas veces lo reemplazaremos utilizando un índice.

Nota: En el último reemplazamos mundo 4 veces por un sólo carácter vacío
>>> cadena = “Hola mundo”
>>> cadena.replace(“o”, “0”)
“H0la mund0”
>>> "Hola mundo mundo mundo mundo mundo".replace(' mundo','',4)
“Hola mundo”

LISTAS

Clear

Como vimos en listas, para “eliminar” todos los elementos de una lista podíamos hacer lista = [], sin embargo, también podemos usar ***clear****()* para vaciar todos los ítems de la lista. Se escribe como: *lista**.clear()***

>>> letras = [‘a’, ‘b’, ‘c’, ‘d’, ‘e’, ‘f’]
>>> letras.clear()
[]

Extend

Como vimos en las listas, podemos sumar una lista con otra lista de la siguiente forma:
>>> numeros = [1,2,3,4]
>>> numeros + [5,6,7,8]
Pero también podemos hacer uso de extend ya que une una lista con otra. Se usa como lista.extend(otra_lista)
>>> lista1 = [1,2,3,4]
>>> lista2 = [5,6,7,8]
>>> lista1.extend(lista2)

Insert

Esta función integrada se usa para agregar un ítem a una lista, pero en un índice específico. Se escribe como: *lista**.insert(**posición, ítem**)**.*

>>> lista = [1,2,3,4,5]
>>> lista.insert(0, 0)
[0,1,2,3,4,5]
>>> lista2 = [5,10,15,25]
>>> lista2.insert(-1, 20) # Anteúltima posición
[5,10,15,20,25]
>>> lista3 = [5,10,15,25]

```
>>> n = len(lista3)
>>> lista3.insert(n, 30) # Última posición
[5,10,15,25,30]
```

Reverse

Esta función integrada sirve para **dar vuelta una lista**. Se escribe como: lista**.reverse()**
Nota: Las cadenas no tienen la función reverse, pero se puede simular haciendo una conversión a lista y después usando el join

```
>>> lista = [1,2,3,4]
>>> lista.reverse()
[4,3,2,1]
```

Sort

Esta función integrada sirve para ordenar una lista automáticamente por valor, de **menor a mayor**. Se escribe como: lista**.sort()**. Si ponemos el argumento **reverse=True** la lista se ordenará de **mayor a menor**.

```
>>> lista = [5,-10,35,0,-65,100
>>> lista.sort()
[-65, -10, 0, 5, 35, 100]
>>> lista.sort(reverse=True)
[100, 35, 5, 0, -10, -65]
```

CONJUNTOS

Copy

Esta función integrada sirve para hacer que se devuelva una **copia** de un set.
Se escribe como: *set**.copy()***

```
>>> set1 = {1,2,3,4}
>>> set2 = set1.copy()
>>> print(set2)
{1,2,3,4}
```

isdisjoint

Esta función comprueba si el set es **distinto** a otro set, es decir, si no hay ningún ítem en común entre ellos. Se escribe como: set1**.isdisjoint(**set2**)**

Nota: Devuelve False por que set1 y set2 comparten el 3

```
>>> set1 = {1,2,3}
>>> set2 = {3,4,5}
>>> set1.isdisjoint(set2)
False
```

issubset

Esta función comprueba si el set es subset de otro set, es decir, si todos sus ítems están en el otro conjunto. Se escribe como: ***set1.issubset(set2)***

Nota: Devuelve False por que set3 no está todo dentro de set4

```
>>> set3 = {-1,99}
>>> set4 = {1,2,3,4,5}
>>> set3.issubset(set4)
False
```

issuperset

Esta función es muy similar al issubset, la diferencia es que esta comprueba si el set es contenedor de otro set, es decir, si contiene todos los ítems de otro set.

Se escribe como: ***set1.issuperset(set2)***

```
>>> set5 = {1,2,3}
>>> set6 = {1,2}
>>> set5.issuperset(set6)
True
```

Unión

Esta función une un set con otro, y devuelve el resultado en un nuevo set. Se escribe como: set1**.union(**set2**)**

```
>>> set1 = {1,2,3}
>>> set2 = {3,4,5}
>>> set1.union(set2)
{1,2,3,4,5}
```

Difference

Esta función encuentra todos los elementos no comunes entre dos set, es decir, nos devuelve un set de ítems diferentes entre cada set.

Se escribe como: set1**.difference(**set2**)**

Nota: Acá devuelve 1 y 2 por qué le pregunta básicamente “que tengo de diferente al set2?”

```
>>> set1 = {1,2,3}
>>> set2 = {3,4,5}
>>> set1.difference(set2)
{1,2}
```

difference_update

Similar al **difference**, pero esta función nos guarda los ítems distintos en el set originales, es decir, le asigna como nuevo valor los ítems diferentes.

Se escribe como: set1**.difference_update(**set2**)**

Nota: Ahora set1 vale {1,2} ya que es la diferencia que tenía con set2

```
>>> set1 = {1,2,3}
>>> set2 = {3,4,5}
>>> set1.difference_update(set2)
>>> print(set1)
{1,2}
```

intersection

Esta función devuelve un set con todos los elementos **comunes** entre dos set, es decir, nos devuelve un set de ítems **iguales** entre cada set.
Se escribe como: set1**.intersection(**set2**)**

```
>>> set1 = {1,2,3}
>>> set2 = {3,4,5}
>>> set1.intersection(set2)
{3}
```

intersection_update

Es exactamente igual al intersection, pero esta función actualiza el set original, es decir, le asigna como nuevo valor los ítems en común. Se escribe como: set1**.intersection_update(**set2**)**

Nota: Ahora set1 vale los ítems en común con set2

```
>>> set1 = {1,2,3}
>>> set2 = {3,4,5}
>>> set1.intersection_update(set2)
>>> print(set1)
{3}
```

DICCIONARIOS

get

La función get sirve para poder buscar un elemento a partir de su **key**, en el caso de no encontrar devuelve un valor por defecto que le indicamos nosotros. Se escribe como: dict**.get(**key, “valor por defecto”**)**

```
>>> colores = { "amarillo":"yellow", "azul":"blue", "verde":"green" }
>>> colores.get(“rojo”, “no hay clave rojo”)
“no hay clave rojo”
>>> colores.get(“amarillo”, “no hay clave amarillo”)
“yellow”
```

keys

La función **key** sirve para poder traer todas las claves de un diccionario en el caso de desconocerlas. Se escribe como: dict**.keys()**

```
>>> colores = { "amarillo":"yellow", "azul":"blue", "verde":"green" }
>>> colores.keys()
dict_keys(['amarillo', 'azul', 'verde'])
```

values

La función **values** es similar a keys, pero esta sirve para poder traer todos los valores de un diccionario. Se escribe como: dict**.values()**

```
>>> colores = { "amarillo":"yellow", "azul":"blue", "verde":"green" }
>>> colores.values()
dict_values(['yellow', 'blue', 'green'])
```

items

La función **items** es similar a keys y values, pero esta crea una lista con clave y valor de los ítems de un diccionario. Se escribe como: dict**.items()**

```
>>> colores = { "amarillo":"yellow", "azul":"blue", "verde":"green" }
>>> colores.items()
dict_items([('amarillo', 'yellow'), ('azul', 'blue'), ('verde', 'green')])

>>> for clave, valor in colores.items():
        print(clave, valor)
amarillo yellow
azul blue
verde green
```