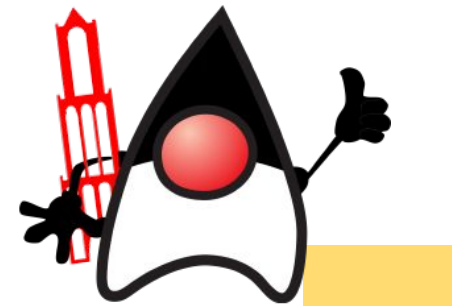
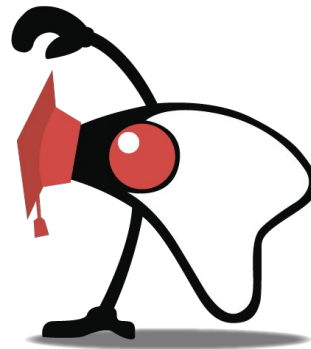
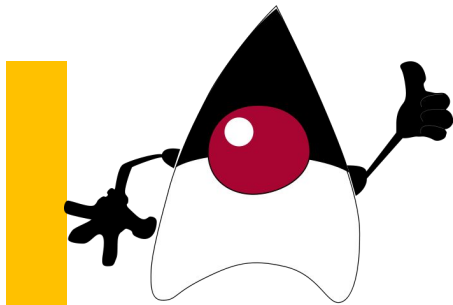




Curso FullStack Python

Codo a Codo 4.0





Python

Parte 3





Cadenas de caracteres

Una **cadena de caracteres (string)** es una secuencia de caracteres. Python soporta caracteres regionales como ñ y vocales con tilde.

Podemos declarar una cadena de caracteres con comillas simples o comillas dobles:

```
cad1= "Lunes"
cad2= 'Martes'
```

PY

Una constante de cadena extensa puede ser distribuida en varias líneas con barra invertida \

```
marca="Alfa Romeo"
frase= 'Dijo "me encanta programar" y comenzó a' \
      "estudiar ingeniería en sistemas"
```

PY

También se permite utilizar otro tipo de comillas en la misma cadena

Las cadenas pueden concatenarse con el operador + (**suma**):

```
nombre= input("Ingrese su nombre: ")
saludo= "Hola "+ nombre
print(saludo) #Hola Pedro
```

PY



Cadenas de caracteres (continuación)

Para concatenar un numero a una cadena, primero se debe convertir a cadena **str(valor)**

```
edad= 25
mensaje= "La edad es: "+ str(edad) # La edad es: 25
altura= 1.75
mensaje= "La altura es: "+ str(altura) # La altura es: 1.75
```

PY

Replicación: Podemos replicar una cadena con *

```
risa='ja'
carcajada= risa*5
print(carcajada) #jajajajaja
asteriscos= "*"*10
print(asteriscos) #*****
```

PY



Cadenas de caracteres (continuación)

Comparación: Se pueden comparar como cualquier otra variable con ==

```
pais= input("Ingrese su pais de nacimiento: ")
if pais == "Argentina":
    print("La nacionalidad es Argentino")
else:
    print("La nacionalidad NO es Argentino")
```

PY

```
Ingrese su pais de nacimiento: argentina
La nacionalidad NO es Argentino
```

terminal

*Es **case sensitive**: distingue entre mayúsculas y minúsculas*

Subíndices: Se pueden acceder usando subíndices:

```
cad="Martes"
print(cad[2])
```

PY

*No se puede modificar usando subíndices, las cadenas **son inmutables**.*

```
r
```

terminal



Cadenas de caracteres (continuación)

Rebanadas: Se pueden manipular usando rebanadas:

```
cad="Miércoles"  
print(cad[1:5])
```

PY

iérc

terminal

Cadenas de caracteres: Funciones

Longitud: Se puede utilizar **len** para saber la longitud:

```
cad="Miércoles"  
print(len(cad))
```

PY

9

terminal



Cadenas de caracteres: Funciones (continuación)

In / Not In: Se pueden utilizar para saber si una subcadena se encuentra o no en una cadena:

```
cad="Jueves"
if "ve" in cad:
    print("se encuentra en la cadena")
else:
    print("no se encuentra en la cadena")

if "iérco1" in cad:
    print("se encuentra en la cadena")
else:
    print("no se encuentra en la cadena")
```

PY

```
se encuentra en la cadena
no se encuentra en la cadena
```

terminal

Cadenas de caracteres: Funciones (continuación)

Iteraciones: Se puede iterar sobre la cadena:

```
cad="Aprendé Python"
for letra in cad:
    print(letra)
```

PY

Max / Min: Se puede utilizar **max()** o **min()**:

```
cad="Python"
print(max(cad)) #y
print(min(cad)) #P
```

PY

*Todas las letras
mayúsculas van antes
de las letras
minúsculas*

Count / index: Se puede utilizar **count()** o **index()**:

```
cad="Programación"
print(cad.count("a")) #2
print(cad.index("grama")) #3
```

PY

A
p
r
e
n
d
é

P
y
t
h
o
n

terminal



Cadenas de caracteres: Métodos

<separador>.join(<secuencia>): Devuelve una cadena con el separador entre cada carácter:

```
cad="12345"  
cad='-' .join(cad)  
print(cad) #1-2-3-4-5
```

PY

<cadena>.split(<sep>): Divide una cadena en una lista, buscando **sep** como separador:

```
cad="Programando en Python"  
lista= cad.split(' ')  
print(lista) #['Programando', 'en', 'Python']
```

PY

<cadena>.replace(<viejo>,<nuevo>,<max>): Reemplaza una cadena por otra hasta un máximo. Si se omite max reemplaza todas las apariciones.

```
cad="Hoy es un día frío. Qué frío está!"  
cad=cad.replace('frío', 'húmedo')  
print(cad) # Hoy es un día húmedo. Qué húmedo está!
```

PY

Cadenas de caracteres: Métodos (continuación)

<cadena>.isalpha(): Devuelve True si **todos** los caracteres de <cadena> son alfabéticos. Sino, devuelve False. Reconoce caracteres regionales.

<cadena>.isdigit(): Devuelve True si **todos** los caracteres de <cadena> son dígitos numéricos.

<cadena>.isalnum(): Devuelve True si **todos** los caracteres de <cadena> son alfabéticos o numéricos.

```
cad="Python"
cad2="Python3"
print(cad.isalpha()) # True
print(cad2.isalpha()) # False
```

```
cad="1234"
cad2="1234a"
print(cad.isdigit()) # True
print(cad2.isdigit()) # False
```

```
cad=""
cad2="12"
cad3="ab"
print(cad.isalnum()) # False
print(cad2.isalnum()) # True
print(cad3.isalnum()) # True
```

Cadenas de caracteres: Métodos (continuación)

<cadena>.isupper(): Devuelve True si **todos** los caracteres de <cadena> están en mayúsculas. Ignora los caracteres no alfabéticos.

<cadena>.islower(): Devuelve True si **todos** los caracteres de <cadena> están en minúsculas. Ignora los caracteres no alfabéticos.

<cadena>.upper(): Devuelve una cadena convertida a mayúsculas. Ignora los caracteres no alfabéticos.

<cadena>.lower(): Devuelve una cadena convertida a minúsculas. Ignora los caracteres no alfabéticos.

```
cad="Python"
cad2="python"
print(cad.isupper()) # False
print(cad2.islower()) # True
print(cad2.upper()) # PYTHON
print(cad.lower()) # python
```

PY

Cadenas de caracteres: Métodos (continuación)

<cadena>.capitalize(): Devuelve una cadena convertida a mayúscula sólo el primer caracter de la primer palabra y el resto en minúsculas.

<cadena>.title(): Devuelve una cadena convertida a mayúscula el primer caracter de cada palabra y el resto en minúsculas.

```
cad="Aprendiendo programación en Python"  
print(cad.capitalize()) # Aprendiendo programación en python  
print(cad.title()) # Aprendiendo Programación En Python
```

PY

<cadena>.center(<ancho>,[relleno]): Devuelve una cadena en el ancho especificado. El resto de la cadena se rellena con espacios o con el caracter relleno.

```
cad1="Hola"  
cad2=cad1.center(10,"*")  
print(cad2) # ***Hola***
```

PY

Cadenas de caracteres: Métodos (continuación)

<cadena>.ljust(<ancho>,[relleno]): Devuelve una cadena alineada a la izquierda en el ancho especificado. El fin de la cadena se rellena con espacios o con el caracter relleno.

<cadena>.rjust(<ancho>,[relleno]): Devuelve una cadena alineada a la derecha en el ancho especificado. El comienzo de la cadena se rellena con espacios o con el caracter relleno.

```
cad1="Hola"  
cad2=cad1.ljust(10, '-')  
cad3=cad1.rjust(10, '-')  
print(cad2) # Hola-----  
print(cad3) # -----Hola
```

PY

<cadena>.zfill(<ancho>): Devuelve una cadena alineada a la derecha en el ancho especificado. El comienzo de la cadena se rellena con ceros.

```
n=3  
cad= str(n).zfill(5)  
print(cad) # 00003
```

PY



Cadenas de caracteres: Métodos (continuación)

<cadena>.lstrip(<str>): Devuelve una cadena sin los caracteres indicados en str al inicio de la cadena.

<cadena>.rstrip(<str>): Devuelve una cadena sin los caracteres indicados en str al final de la cadena.

```
cad="---Hola-Mundo---"  
cad2="---Hola-Mundo---"  
cad= cad.lstrip("-")  
print(cad) #Hola-Mundo---  
cad2= cad2.rstrip("-")  
print(cad2) #---Hola-Mundo
```

PY

<cadena>.strip(<str>): Devuelve una cadena sin los caracteres indicados en str al inicio y al final de la cadena.

```
cad="---Hola-Mundo---"  
cad= cad.strip("-")  
print(cad) #Hola-Mundo
```

PY

Cadenas de caracteres: Métodos (continuación)

<cadena>. find(<str>,[[inicio],[fin]]): Devuelve la posición donde encuentra str en la cadena. Si no lo encuentra devuelve -1 (index devuelve error). Se puede indicar los subíndices desde y hasta a buscar.

<cadena>. rfind(<str>,[[inicio],[fin]]): Similar al find pero busca la última aparición.

```
cad="---Hola-Mundo---"
pos= cad.find("Mundo")
print(pos) #8
cad="---Hola-Mundo--Mundo--"
pos= cad.rfind("Mundo")
print(pos) #15 (última aparición de Mundo)
```

PY



Cadenas de caracteres: Formato

%formato: Disponible desde el inicio de Python.

<str>.format(<datos>): A partir de Python 2.6

```
legajo= 12212
nombre= "María"
nota= 10
#%-formato
print("Legajo: %d Nombre: %s Nota: %d" %(legajo,nombre,nota))
print("Legajo: {} Nombre: {} Nota: {}".format(legajo,nombre,nota))

# En ambos casos devuelve:
# Legajo: 12212 Nombre: María Nota: 10
```

PY

f-string (PEP498): El prefijo f permite dar formato a una cadena de caracteres. Ejemplo:

```
#f-string
print(f"Legajo: {legajo} Nombre: {nombre} Nota: {nota}")
```

PY

El resultado es el mismo que en el caso anterior



Listas

La lista es una secuencia ordenada de elementos. Pueden tener elementos homogéneos del mismo tipo de dato o combinar distintos tipos de datos, aunque por convención en las listas se guardan elementos homogéneos.

Crear listas

Se crean al asignar a una variable una secuencia de elementos encerrados entre corchetes [] y separados por comas. Los corchetes pueden estar juntos, creando una lista vacía.

```
numeros= [1,2,3,4,5] #Lista de números  
dias= ["Lunes", "Martes", "Miércoles"] #Lista de strings  
elementos= [] #Lista vacía
```

PY

Una lista puede contener, a su vez, otra lista:

Lista principal

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

Sublista *Sublista*

PY



Listas | Acceso por subíndice

- El acceso a los elementos se usa el subíndice, el primer elemento siempre lleva el subíndice cero.
- Usar un subíndice negativo hace que la cuenta comience desde atrás.
- Usar subíndices fuera de rango genera un error: *out of range*

5	7	4	4	5	6	2	6	1
0	1	2	3	4	5	6	7	8
-9	-8	-7	-6	-5	-4	-3	-2	-1

Listas | Impresión

- Se pueden imprimir con un ciclo **while** o **for**.
- También pueden imprimirse directamente.

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

PY



Listas | Operador In para iterar listas

- Podemos iterar utilizando el subíndice, utilizando un contador, **while** y **len** para generar la secuencia de índices:

```
#Mostrar la lista, separando los elementos con un espacio
def MostrarLista(lista):
    i = 0
    while i < len(lista):
        print(lista[i], end=" ")
        i = i + 1
    print()

#Declaración de la lista y llamado a la función
lista=["A", "B", "C", "D", "E"]
MostrarLista(lista)
```

PY

A B C D E

terminal



Listas | Operador In para iterar listas

- Podemos iterar utilizando el subíndice, for y range para generar la secuencia de índices:

```
def SumarLista(lista):  
    suma = 0  
    for i in range(len(lista)):  
        suma = suma + lista[i]  
    return suma  
  
#Declaración de la lista y llamado a la función  
lista=[2,3,4,5,6]  
print(SumarLista(lista))
```

PY

20

terminal

- Podemos iterar en forma directa los elementos de la lista, sin necesidad de generar la secuencia de subíndices. En este caso la variable *i* toma el elemento de la lista.

```
vocales=['a','e','i','o','u']  
for i in vocales:  
    print(i, end="-")
```

PY

a-e-i-o-u-

terminal



Listas | Desempaquetado

- Consiste en asignar sus elementos a un conjunto de variables:

```
dias= ["Lunes", "Martes", "Miércoles"]
d1, d2, d3 = dias
print(d1)
print(d2)
print(d3)
```

PY

```
Lunes
Martes
Miércoles
```

terminal

Listas | Operaciones

- Las listas pueden **concatenarse** con el operador **suma**:

```
lista1= [1,2,3]
lista2= [4,5,6]
lista3= lista1 + lista2
print(lista3) #[1,2,3,4,5,6]
```

PY





Listas | Operaciones (continuación)

- La concatenación nos permite **agregar** elementos nuevos a la lista:

```
lista=[3,4,5]  
lista= lista + [6] #[3,4,5,6]
```

PY

El elemento debe encerrarse entre corchetes para que sea una lista.

- Podemos **modificar** los elementos de una lista utilizando el subíndice:

```
lista=[3,4,5]  
lista[1]=7  
print(lista) #[3,7,5]
```

PY

- La función **len()** devuelve la cantidad de elementos de una lista:

```
lista=[3,4,5,6]  
print(len(lista)) # 4
```

PY



Listas | Operaciones (continuación)

- La función **max()** devuelve el mayor elemento de una lista.
- La función **min()** devuelve el menor elemento de una lista.
- La función **sum()** devuelve la suma de los elementos de una lista:

```
lista=[3,4,5,6]
print(max(lista)) # 6
print(min(lista)) # 3
print(sum(lista)) # 18
```

PY

Para utilizar **max()**, **min()**
y **sum()** la lista debe ser
homogénea

- La función **list()** convierte cualquier secuencia a una lista. Se puede utilizar con rangos, cadenas y algunos más...

```
lista= list(range(6))
print(lista) #[0,1,2,3,4,5]
```

PY

- El operador **in** permite verificar la presencia de un elemento, mientras que la ausencia de un elemento se comprueba con **not in**:

```
lista=[3,4,5,6]
print(4 in lista) #True
print(8 in lista) #False
```

PY

Devuelven True o False



Listas | Métodos

Un **método** es un procedimiento o función que pertenece a un objeto. Permiten manipular los datos almacenados en él y se escriben luego del nombre del objeto, separados por un punto.

- **append():** Agrega un elemento al final de la lista.

```
lista=[3,4,5]
lista.append(6)
print(lista) #[3,4,5,6]
```

PY

- **insert(<pos>, <elemento>):** Inserta un elemento en una posición determinada de la lista.

```
lista=[3,4,5]
lista.insert(0,2) #Agrega en la posición 0 el número 2
print(lista) #[2,3,4,5]
lista.insert(3,25) #Agrega en la posición 3 el número 25
print(lista) #[2,3,4,25,5]
```

PY



Listas | Métodos (continuación)

- **pop():** Elimina el último elemento de la lista.

```
lista=[6,9,8]  
lista.pop() # queda [6,9]
```

PY

- **pop(<posición>):** Elimina un elemento en una posición determinada de la lista.

```
lista=[3,4,5]  
lista.pop(1) # queda [3,5]
```

PY

*Provoca un error
si no existe*

- **remove(<valor>):** Elimina un elemento en la lista, identificado por su valor.

```
lista=[3,4,5]  
lista.remove(3) # queda [4,5]
```

PY

*Provoca un error si no existe.
Si está más de una vez
elimina la primera aparición*

- **index(<valor>):** Busca un valor y devuelve su posición.

```
lista=[3,4,5]  
print(lista.index(5)) #2
```

PY

*Provoca un error si no existe.
Si está más de una vez devuelve
la primera aparición*

El método admite como argumento adicional un índice inicial a partir de donde comenzar la búsqueda (`lista.index(5,2)`), opcionalmente también el índice final (`lista.index(5,2,4)`).



Listas | Métodos (continuación)

- **count():** Devuelve la cantidad de repeticiones de un elemento, cero si no lo encuentra.

```
lista=[3,4,5,3,5,8,5]
print(lista.count(5)) #El número 5 está 3 veces
print(lista.count(2)) #El número 2 no está en la lista
```

PY

- **reverse():** Invierte el orden de los elementos de una lista.

```
lista=[3,4,5]
lista.reverse()
print(lista) #[5,4,3]
```

PY





Listas | Métodos (continuación)

- **sort ()**: Ordena los elementos de la lista.

```
lista=[5, 1, 7, 2]  
lista.sort()  
print(lista) #[1,2,5,7]
```

PY

- **sort (reverse=True)**: Con parámetro reverse en True, ordena la lista de mayor a menor.

```
lista=[5, 1, 7, 2]  
lista.sort(reverse=True)  
print(lista) #[7,5,2,1]
```

PY

- **clear()**: Elimina todos los elementos de una lista.

```
lista=[3,4,5]  
lista.clear()  
print(lista) #[]
```

PY





Diccionarios

Conjunto no ordenado de pares *-clave: valor-*. Las claves son únicas. Si se quiere guardar un valor a una clave ya existente se pierde dicho valor. Las claves pueden ser cualquier elemento de tipo inmutable (cadenas, números o tuplas si estas sólo contienen cadenas, números o tuplas). Se representa como una lista de pares clave:valor separados por comas encerrados entre llaves.

Para acceder a las claves Python utiliza un método de hash. Ejemplo:

```
{}
```

```
{'Juan': 56}
```

```
{'Juan': 56, 'Ana': 15}
```

```
# diccionario vacío
```

```
# diccionario de un elemento
```

```
# diccionario de dos elementos
```

Creación:

- Por extensión. Los elementos se enumeran.
- Por compresión.

```
# Creación: Por extensión
```

```
diccionario = {'Juan': 56, 'Ana': 15}
```

```
# Creación: Por compresión
```

```
diccionario = {x: x ** 2 for x in (2, 4, 6)}
```

Diccionarios: Accesos

Puede accederse:

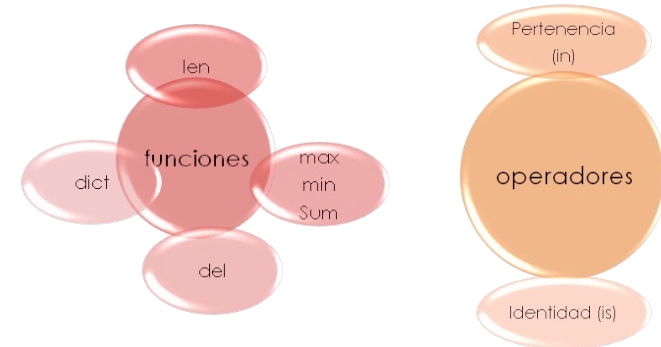
- A las claves, utilizando método `keys()`
- A los valores, utilizando la clave como índice
- A la clave-valor, utilizando método `items()`
- No es posible obtener porciones de un diccionario usando `[:]`, ya que las claves no tienen.

```
diccionario = {1: 'uno', 2: 'dos', 3: 'tres'}  
print(diccionario.keys())  
for i in diccionario.keys():  
    print(diccionario[i])  
for clave, valor in diccionario.items():  
    print(clave, ': ', valor, end= ' ; ')
```

PY

```
dict_keys([1, 2, 3])  
uno  
dos  
tres  
1 : uno; 2 : dos; 3 : tres;
```

terminal





Tuplas

Conjunto de elementos separados por comas y encerrados entre paréntesis. Los paréntesis no son obligatorios. Estas son **inmutables** y en general contienen una secuencia heterogénea de elementos. Los elementos pueden ser mutables. Ejemplos:

```
( ) # tupla vacía
'un valor', # tupla con un valor
('uno', 'dos', 'tres') # cadenas
('Palotes, Juan de', (1930, 11, 13), 3000936) # datos de persona
'Palotes, Juan de', (1930, 11, 13), 3000936 # datos de persona
```

Creación: Por extensión. Los elementos se enumeran. Ejemplo:

```
tupla = ('uno', 'dos', 'tres')
```

Empaquetado (zip): Se asigna a la tupla una enumeración de variables o valores. Ejemplo:

```
tupla = 'Palotes, Juan de', (1930, 11, 13), 3000936
```

Desempaquetado (unpack): Se asigna los valores de una tupla a un conjunto de variables. una enumeración de variables o valores. requiere que la cantidad de variables a la izquierda del signo igual sea el tamaño de la tupla. Ejemplo:

```
nombre, nacimiento, dni =( 'Palotes, Juan de', (1930, 11, 13), 3000936)
```

Tuplas: Accesos

Puede accederse:

- Desempaquetando
- Con un índice, comenzando de la posición 0
- Por método rebanada o *Slicing* `tupla[comienzo:fin:salto]`

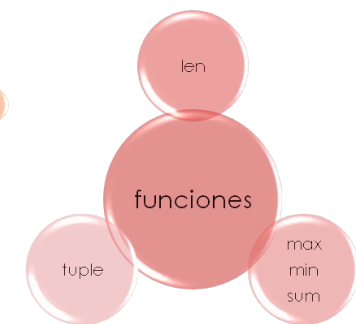
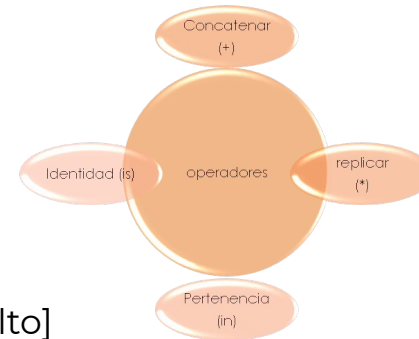
Ejemplos:

```
tupla = ('Palotes, Juan de', (1930, 11, 13), 3000936)
nombre, fecha, dni = tupla
print('Nombre: ', nombre, '. Fecha nac.: ', fecha, '. DNI: ', dni)
print('Nombre: ', tupla[0], '. Fecha nac.: ', tupla[1], '. DNI: ', tupla[2])
print(tupla[::])
```

```
Nombre: Palotes, Juan de . Fecha nac.: (1930, 11, 13) . DNI: 3000936
Nombre: Palotes, Juan de . Fecha nac.: (1930, 11, 13) . DNI: 3000936
('Palotes, Juan de', (1930, 11, 13), 3000936)
```

PY

terminal



Conjuntos

Conjunto de elementos únicos y no ordenados separados por comas y encerrados por llaves. Los elementos no pueden ser mutables. Ejemplos:

```
set() # conjunto vacío
{'un valor'} # conjunto con un valor
{'uno', 'dos', 'tres'} # conjunto de cadenas
{'Palotes, Juan de', (1930, 11, 13), 3000936} # datos de persona
```

Creación: Por extensión. Los elementos se enumeran. Ejemplo:

```
conjunto = { 1, 9, 0 }
```

Accesos: Puede accederse:

- Iterando
- Utilizando el método pop
- No puede accederse a través de un subíndice pues sus elementos no están ordenados

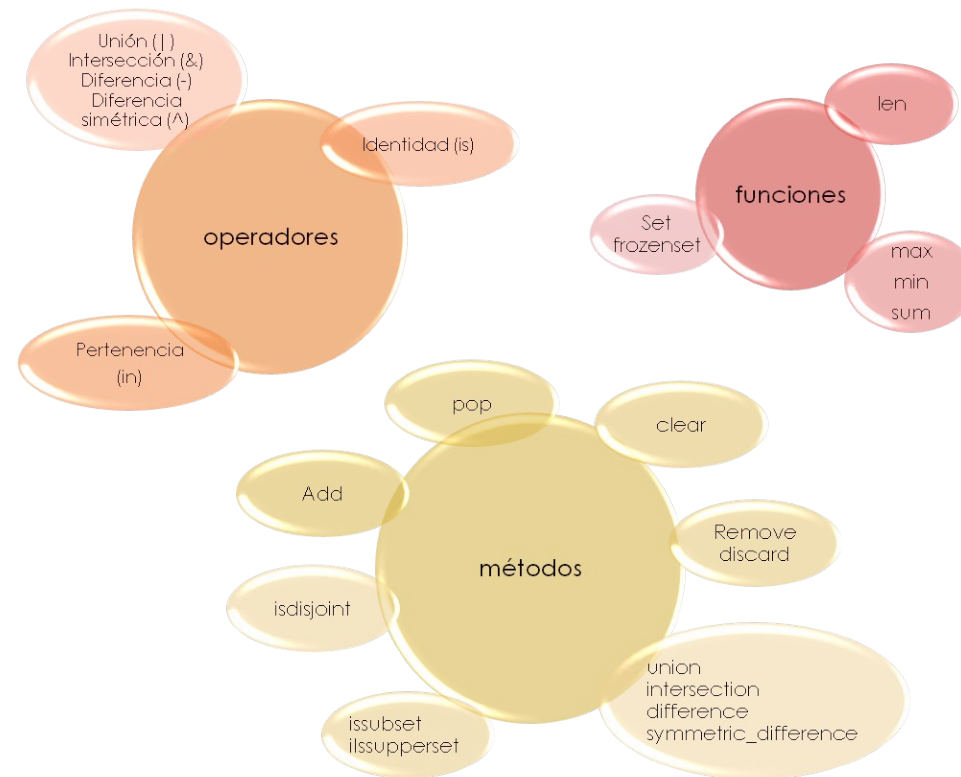
```
conjunto = {'Palotes, Juan de', (1930, 11, 13), 3000936}
a = set(conjunto)
print(a)
[print (elem) for elem in a]
print(a)
```

```
{3000936, 'Palotes, Juan de', (1930, 11, 13)}
3000936
Palotes, Juan de
(1930, 11, 13)
{3000936, 'Palotes, Juan de', (1930, 11, 13)}
```

terminal

PY

Conjuntos





Material complementario

- **Videos tutoriales:**

- **Tuplas:**

- <https://www.youtube.com/watch?v=Ufqh8aoR9hE&list=PLU8oAlHdN5BlvPxziopYZRd55pdqFwkeS&index=8>

- **Diccionarios:**

- <https://www.youtube.com/watch?v=2OmgHI8lp0I&list=PLU8oAlHdN5BlvPxziopYZRd55pdqFwkeS&index=9>

- **Conjuntos:**

- <https://www.youtube.com/watch?v=rmRrvol4XcM&list=PLWtYZ2ejMVJnh0KVllw24XklzJ62WNFs&index=28> (parte 1)
 - <https://www.youtube.com/watch?v=UKD3CMINxik&list=PLWtYZ2ejMVJnh0KVllw24XklzJ62WNFs&index=29> (parte 2)

- **w3schools:**

- **Diccionarios:** https://www.w3schools.com/python/python_dictionaries.asp
 - **Tuplas:** https://www.w3schools.com/python/python_tuples.asp
 - **Conjuntos:** https://www.w3schools.com/python/python_sets.asp