

booleanos

Los booleanos son tipos de datos binarios ([True](#)/[False](#)), que surgen de operaciones lógicas, o pueden declararse explícitamente.

operadores lógicos

`==` igual a

`!=` diferente a

`>` mayor que

`<` menor que

`>=` mayor o igual que

`<=` menor o igual que

`and` y ([True](#) si dos declaraciones son [True](#))

`or` o ([True](#) si al menos una declaración es [True](#))

`not` no (invierte el valor del booleano)

sets

Los sets son otro tipo de estructuras de datos. Se diferencian de listas, tuplas y diccionarios porque son una colección *mutable* de elementos *inmutables*, *no ordenados* y *sin datos repetidos*.

mutable ✓

ordenado ✗

^{admite}
duplicados ✗

```
mi_set_a = {1, 2, "tres"}    mi_set_b = {3, "tres"}
```

métodos

add(item) agrega un elemento al set

```
mi_set_a.add(5)
print(mi_set_a)
>> {1, 2, "tres", 5}
```

clear() remueve todos los elementos de un set

```
mi_set_a.clear()
print(mi_set_a)
>> set()
```

copy() retorna una copia del set

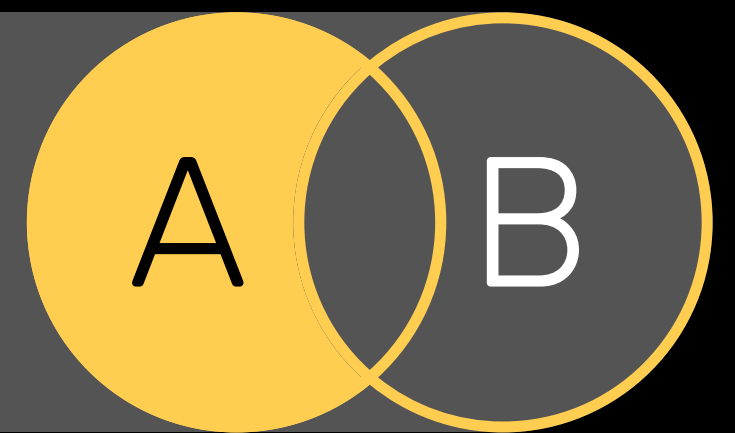
```
mi_set_c = mi_set_a.copy()
print(mi_set_c)
>> {1, 2, "tres"}
```

sets

```
mi_set_a = {1, 2, "tres"}    mi_set_b = {3, "tres"}
```

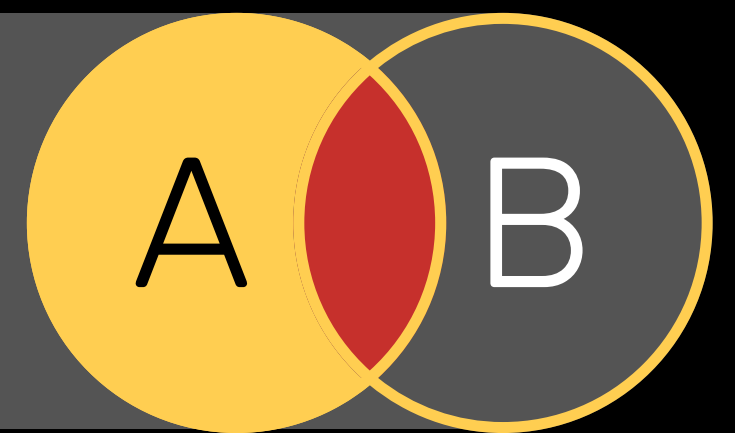
difference(set) retorna el set formado por todos los elementos que únicamente existen en el set A

```
mi_set_c = mi_set_a.difference(mi_set_b)
print(mi_set_c)
>> {1, 2}
```



difference_update(set) remueve de A todos los elementos comunes a A y B

```
mi_set_a.difference_update(mi_set_b)
print(mi_set_a)
>> {1, 2}
```

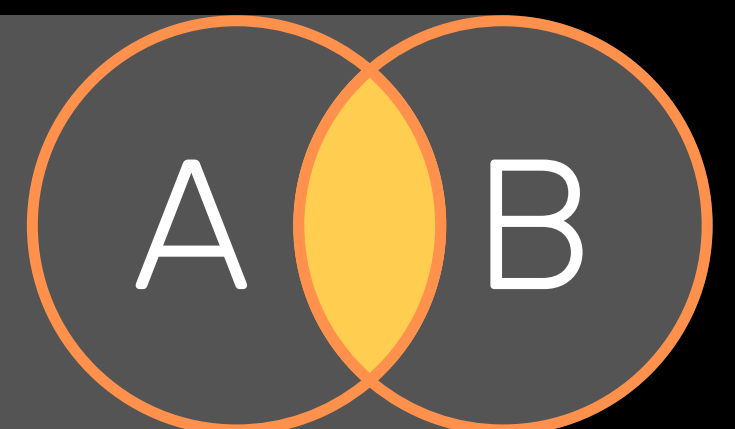


discard(item) remueve un elemento del set

```
mi_set_a.discard("tres")
print(mi_set_a)
>> {1, 2}
```

intersection(set) retorna el set formado por todos los elementos que existen en A y B simultáneamente.

```
mi_set_c = mi_set_a.intersection(mi_set_b)
print(mi_set_c)
>> {'tres'}
```

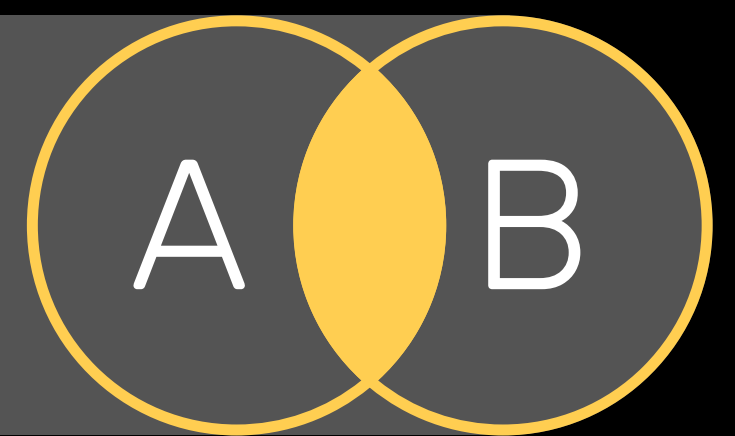


sets

```
mi_set_a = {1, 2, "tres"}    mi_set_b = {3, "tres"}
```

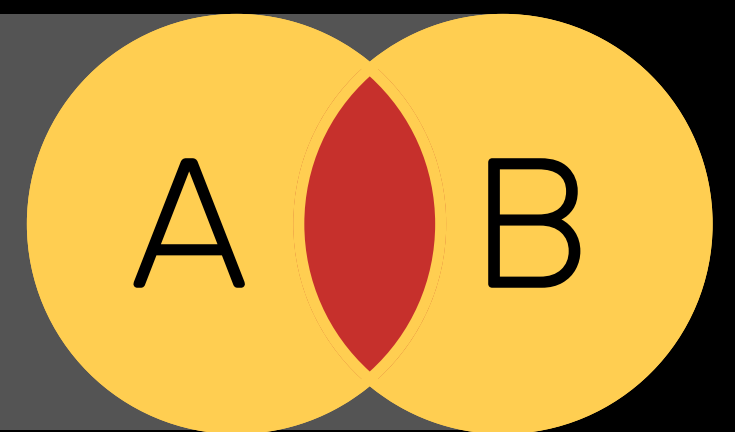
intersection_update(set) mantiene únicamente los elementos comunes a A y B

```
mi_set_b.intersection_update(mi_set_a)
print(mi_set_b)
>> {"tres"}
```



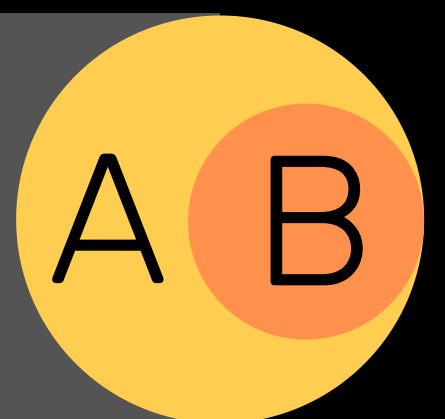
isdisjoint(set) devuelve True si A y B no tienen elementos en común

```
conjunto_disjunto = mi_set_a.isdisjoint(mi_set_b)
print(conjunto_disjunto)
>> False
```



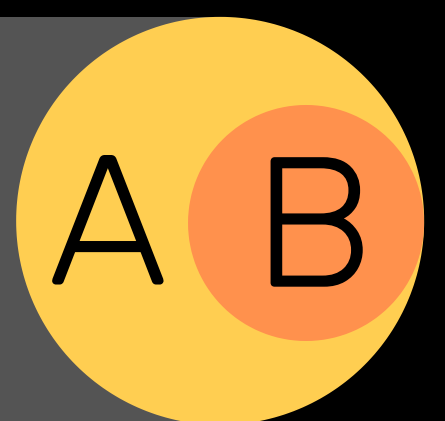
issubset(set) devuelve True si todos los elementos de B están presentes en A

```
es_subset = mi_set_b.issubset(mi_set_a)
print(es_subset)
>> False
```



issuperset(set) devuelve True si A contiene todos los elementos de B

```
es_superset = mi_set_a.issuperset(mi_set_b)
print(es_superset)
>> False
```



sets

```
mi_set_a = {1, 2, "tres"}    mi_set_b = {3, "tres"}
```

pop() elimina y retorna un elemento al azar del set

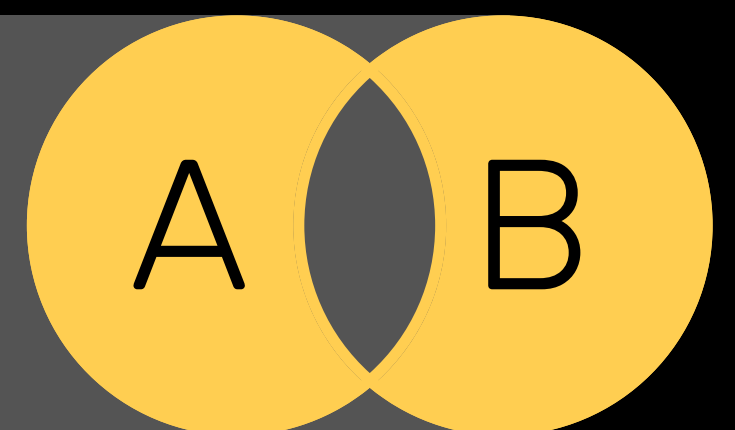
```
aleatorio = mi_set_a.pop()  
print(aleatorio)  
>> {2}
```

remove(item) elimina un item del set, y arroja error si no existe

```
mi_set_a.remove("tres")  
print(mi_set_a)  
>> {1, 2}
```

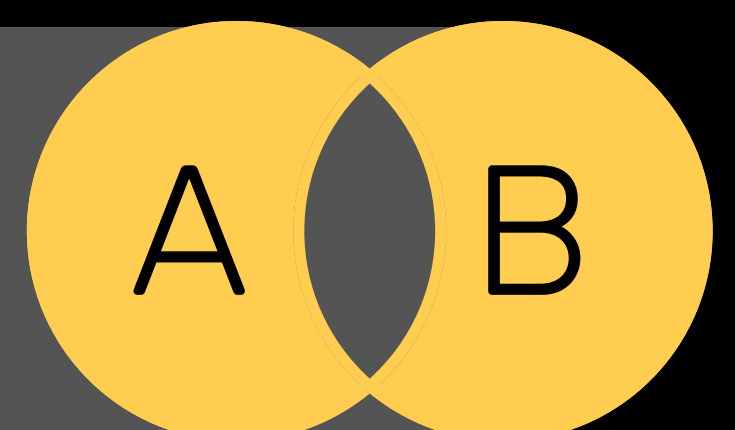
symmetric_difference(set) retorna todos los elementos de A y B, excepto aquellos que son comunes a los dos

```
mi_set_c = mi_set_b.symmetric_difference(mi_set_a)  
print(mi_set_c)  
>> {1, 2, 3}
```



symmetric_difference_update(set) elimina los elementos comunes a A y B, agregando los que no están presentes en ambos a la vez

```
mi_set_b.symmetric_difference_update(mi_set_a)  
print(mi_set_b)  
>> {1, 2, 3}
```

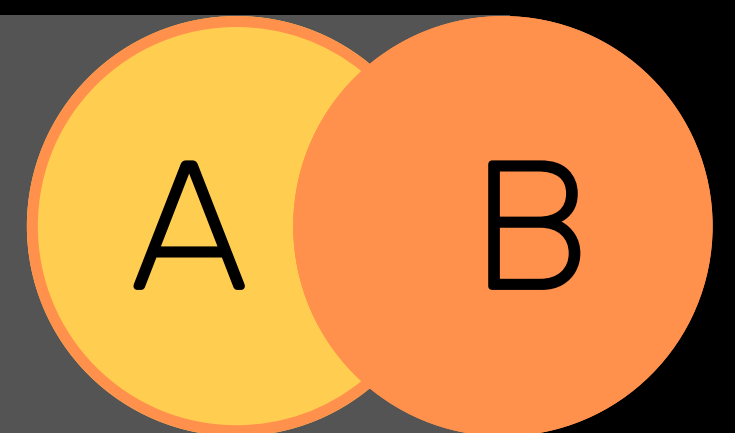


sets

```
mi_set_a = {1, 2, "tres"}    mi_set_b = {3, "tres"}
```

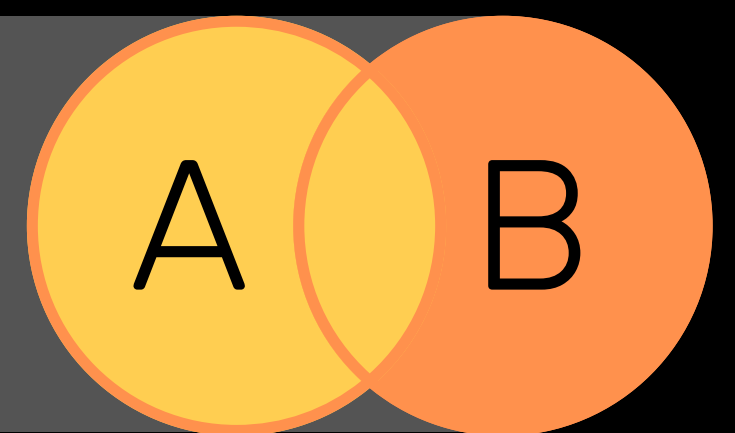
union(set) retorna un set resultado de combinar A y B (los datos duplicados se eliminan)

```
mi_set_c = mi_set_a.union(mi_set_b)
print(mi_set_c)
>> {1, 2, 3, 'tres'}
```



update(set) inserta en A los elementos de B

```
mi_set_a.update(mi_set_b)
print(mi_set_a)
>> {1, 2, 3, 'tres'}
```



tuples

Los tuples o *tuplas*, son estructuras de datos que almacenan múltiples elementos en una única variable. Se caracterizan por ser ordenadas e *inmutables*. Esta característica las hace más eficientes en memoria y a prueba de daños.

mutable ✗ ordenado ✓ admite duplicados ✓

```
mi_tuple = (1, "dos", [3.33, "cuatro"], (5.0, 6))
```

indexado (acceso a datos)

```
print(mi_tuple[3][0])  
>> 5.0
```

casting (conversión de tipos de datos)

```
lista_1 = list(mi_tuple)  
print(lista_1)  
>> [1, "dos", [3.33, "cuatro"], (5.0, 6)]
```

ahora es una estructura mutable ↗

unpacking (extracción de datos)

```
a, b, c, d = mi_tuple  
print(c)  
>> [3.33, "cuatro"]
```

diccionarios

Los diccionarios son estructuras de datos que almacenan información en pares **clave:valor**. Son especialmente útiles para guardar y recuperar información a partir de los nombres de sus claves (no utilizan índices).

mutable ✓

ordenado ✗ *

admite duplicados ✗: ✓
valor
clave

```
mi_diccionario = {"curso": "Python TOTAL", "clase": "Diccionarios"}
```

agregar nuevos datos, o modificarlos

```
mi_diccionario["recursos"] = ["notas", "ejercicios"]
```

acceso a valores a través del nombre de las claves

```
print(mi_diccionario["recursos"][1])  
>> "ejercicios"
```

métodos para listar los nombres de las **claves**, **valores**, y pares **clave:valor**

keys() ← values() → items()

**: A partir de Python 3.7+, los diccionarios son tipos de datos ordenados, en el sentido que dicho orden se mantiene según su orden de inserción para aumentar la eficiencia en el uso de la memoria.*

listas

Las listas son secuencias ordenadas de objetos. Estos objetos pueden ser datos de cualquier tipo: strings, integers, floats, booleanos, listas, entre otros. Son tipos de datos mutables.

mutable ✓

ordenado ✓

^{admite}
duplicados ✓

```
lista_1 = ["C", "C++", "Python", "Java"]  
lista_2 = ["PHP", "SQL", "Visual Basic"]
```

indexado: podemos acceder a los elementos de una lista a través de sus índices [inicio:fin:paso]

```
print(lista_1[1:3])  
>> ["C++", "Python"]
```

cantidad de elementos: a través de la propiedad len()

```
print(len(lista_1))  
>> 4
```

concatenación: sumamos los elementos de varias listas con el símbolo +

```
print(lista_1 + lista_2)  
>> ['C', 'C++', 'Python', 'Java', 'PHP', 'SQL', 'Visual Basic']
```

listas

```
lista_1 = ["C", "C++", "Python", "Java"]
lista_2 = ["PHP", "SQL", "Visual Basic"]
lista_3 = ["d", "a", "c", "b", "e"]
lista_4 = [5, 4, 7, 1, 9]
```

función `append()`: agrega un elemento a una lista *en el lugar*

```
lista_1.append("R")
print(lista_1)
>> ["C", "C++", "Python", "Java", "R"]
```

función `pop()`: elimina un elemento de la lista dado el índice, y *devuelve* el valor quitado

```
print(lista_1.pop(4))
>> "R"
```

función `sort()`: ordena los elementos de la lista *en el lugar*

```
lista_3.sort()
print(lista_3)
>> ['a', 'b', 'c', 'd', 'e']
```

función `reverse()`: invierte el orden de los elementos *en el lugar*

```
lista_4.reverse()
print(lista_4)
>> [9, 1, 7, 4, 5]
```

➡ `reverse` no es lo opuesto a `sort`

strings: propiedades

Esto es lo que debes tener presente al trabajar con strings en Python:

- **son inmutables:** una vez creados, no pueden modificarse sus partes, pero sí pueden reasignarse los valores de las variables a través de métodos de strings
- **concatenable:** es posible unir strings con el símbolo +
- **multiplicable:** es posible concatenar repeticiones de un string con el símbolo *
- **multilineales:** pueden escribirse en varias líneas al encerrarse entre triples comillas simples (''' ''') o dobles (""" """)
- **determinar su longitud:** a través de la función `len(mi_string)`
- **verificar su contenido:** a través de las palabras clave `in` y `not in`. El resultado de esta verificación es un booleano (`True` / `False`).

strings: métodos

métodos de análisis

count() : retorna el número de veces que se repite un conjunto de caracteres especificado.

```
"Hola mundo".count("Hola")  
>> 1
```

find() e **index()** retornan la ubicación (comenzando desde el cero) en la que se encuentra el argumento indicado. Difieren en que **index** lanza **ValueError** cuando el argumento no es encontrado, mientras **find** retorna **-1**.

```
"Hola mundo".find("world")  
>> -1
```

rfind() y **rindex()**. Para buscar un conjunto de caracteres pero desde el final.

```
"C:/python36/python.exe".rfind("/")  
>> 11
```

startswith() y **endswith()** indican si la cadena en cuestión comienza o termina con el conjunto de caracteres pasados como argumento, y retornan **True** o **False** en función de ello.

```
"Hola mundo".startswith("Hola")  
>> True
```


strings: métodos

métodos de análisis

isdigit(): determina si todos los caracteres de la cadena son **dígitos**, o **pueden formar números**, incluidos aquellos correspondientes a lenguas orientales.

```
"abc123".isdigit()  
>> False
```

isnumeric(): determina si todos los caracteres de la cadena son **números**, incluye **también caracteres de connotación numérica** que no necesariamente son dígitos (por ejemplo, una fracción).

```
"1234".isnumeric()  
>> True
```

isdecimal(): determina si todos los caracteres de la cadena son **decimales**; esto es, formados por dígitos **del 0 al 9**.

```
"1234".isdecimal()  
>> True
```

isalnum(): determina si todos los caracteres de la cadena son **alfanuméricos**.

```
"abc123".isalnum()  
>> True
```

isalpha(): determina si todos los caracteres de la cadena son **alfabéticos**.

```
"abc123".isalpha()  
>> False
```


strings: métodos

métodos de análisis

islower(): determina si todos los caracteres de la cadena son **minúsculas**.

```
"abcdef".islower()  
>> True
```

isupper(): determina si todos los caracteres de la cadena son **mayúsculas**.

```
"ABCDEF".isupper()  
>> True
```

isprintable(): determina si todos los caracteres de la cadena son **imprimibles** (es decir, no son caracteres especiales indicados por \...).

```
"Hola \t mundo!".isprintable()  
>> False
```

isspace(): determina si todos los caracteres de la cadena son **espacios**.

```
"Hola mundo".isspace()  
>> False
```

strings: métodos

métodos de transformación

En realidad los strings son inmutables; por ende, todos los métodos a continuación no actúan sobre el objeto original sino que retornan uno nuevo.

capitalize() retorna la cadena con su primera letra en mayúscula.

```
"hola mundo".capitalize()  
>> 'Hola mundo'
```

encode() codifica la cadena con el mapa de caracteres especificado y retorna una instancia del tipo bytes.

```
"Hola mundo".encode("utf-8")  
>> b'Hola mundo'
```

replace() reemplaza una cadena por otra.

```
"Hola mundo".replace("mundo", "world")  
>> 'Hola world'
```

lower() retorna una copia de la cadena con todas sus letras en minúsculas.

```
"Hola Mundo!".lower()  
>> 'hola mundo!'
```

upper() retorna una copia de la cadena con todas sus letras en mayúsculas.

```
"Hola Mundo!".upper()  
>> 'HOLA MUNDO!'
```

strings: métodos

métodos de transformación

swapcase() cambia las mayúsculas por minúsculas y viceversa.

```
"Hola Mundo!".swapcase()  
>> 'hOLA mUNDO!'
```

strip(), **lstrip()** y **rstrip()** remueven los espacios en blanco que preceden y/o suceden a la cadena.

```
"  Hola mundo!  ".strip()  
>> 'Hola mundo!'
```

Los métodos **center()**, **ljust()** y **rjust()** alinean una cadena en el centro, la izquierda o la derecha. Un segundo argumento indica con qué carácter se deben llenar los espacios vacíos (por defecto un espacio en blanco).

```
"Hola".center(10, "*")  
>> '***Hola***'
```

métodos de separación y unión

split() divide una cadena según un carácter separador (por defecto son espacios en blanco). Un segundo argumento en **split()** indica cuál es el máximo de divisiones que puede tener lugar (-1 por defecto para representar una cantidad ilimitada).

```
"Hola mundo!\nHello world!".split()  
>> ['Hola', 'mundo!', 'Hello', 'world!']
```

strings: métodos

métodos de separación y unión

splitlines() divide una cadena con cada aparición de un salto de línea.

```
"Hola mundo!\nHello world!".splitlines()  
>> ['Hola mundo!', 'Hello world!']
```

partition() retorna una tupla de tres elementos: el bloque de caracteres anterior a la primera ocurrencia del separador, el separador mismo, y el bloque posterior.

```
"Hola mundo. Hello world!".partition(" ")  
>> ('Hola', ' ', 'mundo. Hello world!')
```

rpartition() opera del mismo modo que el anterior, pero partiendo de derecha a izquierda.

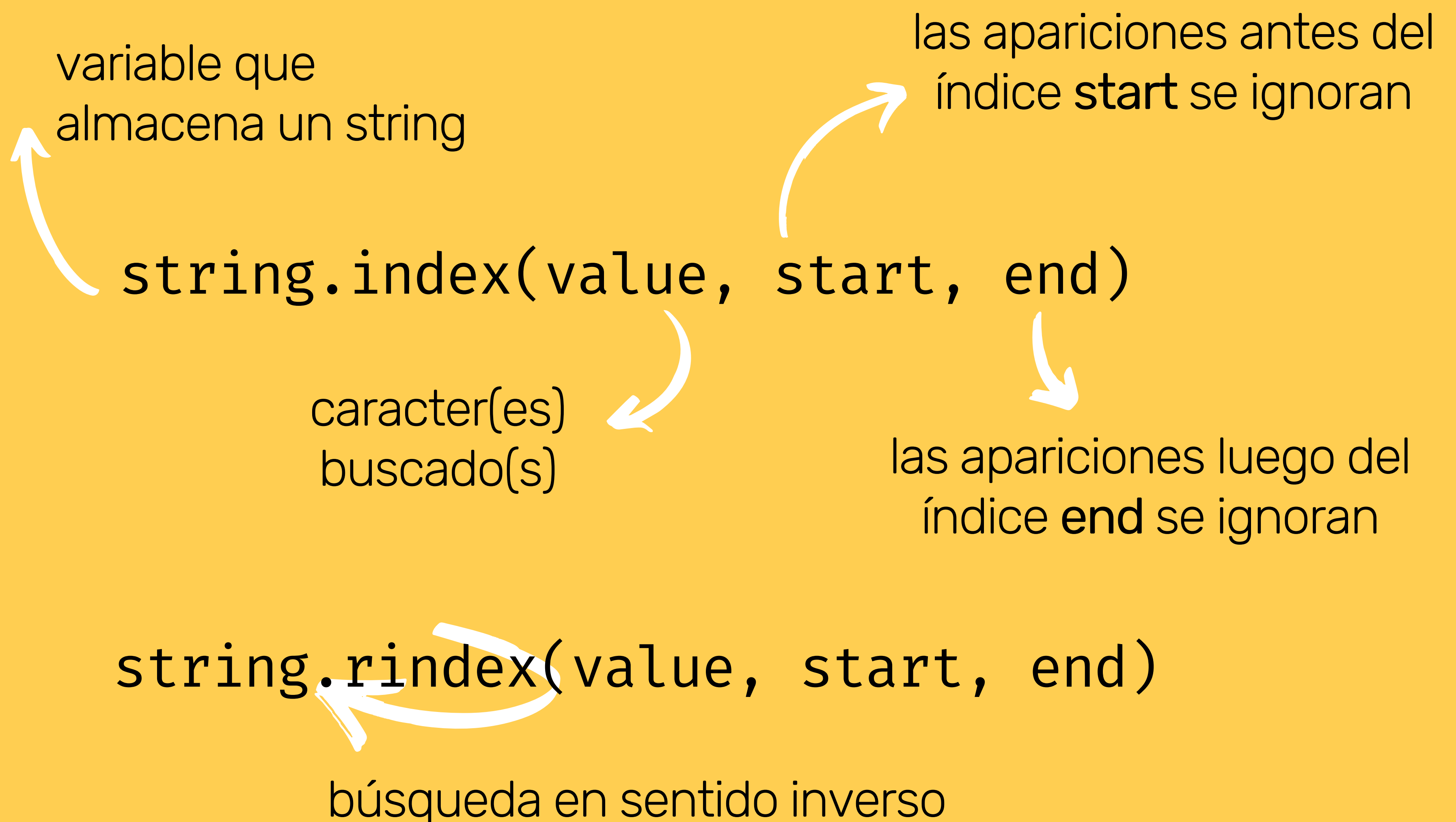
```
"Hola mundo. Hello world!".rpartition(" ")  
>> ('Hola mundo. Hello', ' ', 'world!')
```

join() debe ser llamado desde una cadena que actúa como separador para unir dentro de una misma cadena resultante los elementos de una lista.

```
", ".join(["C", "C++", "Python", "Java"])  
>> 'C, C++, Python, Java'
```


index()

Utilizamos el método `index()` para explorar strings, ya que permite hallar el índice de aparición de un carácter o cadena de caracteres dentro de un texto dado.



`string[i]` → devuelve el carácter en el índice `i`*

**: En Python, el índice en primera posición es el 0*

substrings

Podemos extraer porciones de texto utilizando las herramientas de manipulación de strings conocidas como slicing (*rebanar*).

índice de inicio del sub-string
(incluido)

string[start:stop:step]

paso

índice del final del sub-string
(no incluido)

saludo = H o l a _ M u n d o

```
print(saludo[2:6])
```

```
>> la_M
```

H o l a _ M u n d o

0 1 2 3 4 5 6 7 8 9

```
print(saludo[3::3])
```

```
>> auo
```

H o l a _ M u n d o

0 1 2 3 4 5 6 7 8 9

```
print(saludo[::-1])
```

```
>> odnuM_aloH
```

H o l a _ M u n d o

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