

print

Declaración que al ejecutarse muestra (o imprime) en pantalla el argumento que se introduce dentro de los paréntesis.

mostrar texto

Ingresamos entre comillas simples o dobles los caracteres de texto que deben mostrarse en pantalla.

```
print("Hola mundo")
```

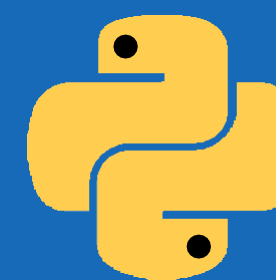
```
>> Hola mundo
```

mostrar números

Podemos entregarle a print() el número que debe mostrar, o una operación matemática a resolver. No empleamos comillas en estos casos.

```
print(150 + 50)
```

```
>> 200
```



strings

Los strings en Python son un tipo de dato formado por cadenas (o secuencias) de caracteres de cualquier tipo, formando un texto.

concatenación

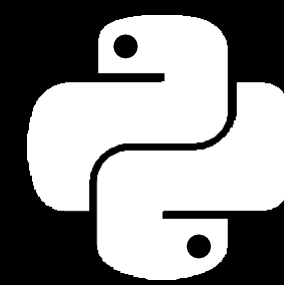
Unificación de cadenas de texto:

```
print("Hola" + " " + "mundo")  
  
>> Hola mundo
```

caracteres especiales

Indicamos a la consola que el caracter a continuación del símbolo \ debe ser tratado como un caracter especial.

```
\ " > Imprime comillas  
\n > Separa texto en una nueva línea  
\t > Imprime un tabulador  
\ > Imprime la barra invertida textualmente
```

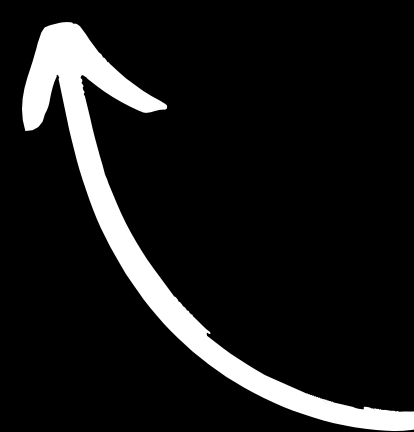


input

Función que permite al usuario introducir información por medio del teclado al ejecutarse, otorgándole una instrucción acerca del ingreso solicitado. El código continuará ejecutándose luego de que el usuario realice la acción.

```
input("Dime tu nombre: ")
```

```
>> Dime tu nombre: |
```



Ingreso por teclado

```
print("Tu nombre es " + input("Dime tu  
nombre: ") )
```

```
>> Dime tu nombre: Federico
```

```
>> Tu nombre es Federico
```

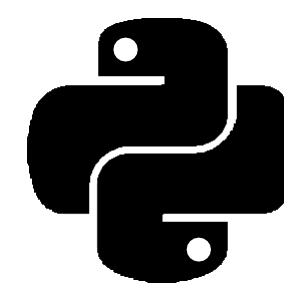
Ejercicio 1

Imagina esta situación: tu mejor amigo ha puesto una fábrica de cerveza y tiene todo listo. Su producto es fantástico, tiene cuerpo, buen sabor, buen color, el nivel justo de espuma... pero le falta una identidad. No se le ocurre qué nombre ponerle su cerveza para que tenga una identidad única y original. Entonces vienes tú y le dices "No te preocupes, yo voy a crear un programa que te va a hacer dos preguntas y luego te va a decir cuál es el nombre de tu cerveza". Así de simple.

Ya sé que en el mundo real no necesitaríamos desarrollar un software solo para hacer dos preguntas, pero hasta que aprendamos más funcionalidades los programas van a tener que mantenerse en el terreno de lo simple. Igualmente, si está recién comenzando, este va a ser todo un desafío.

Vas a crear un código en Python que le pida a tu amigo que responda dos preguntas que requieran una sola palabra cada una y que luego le muestre en pantalla esas palabras combinadas, para formar una marca creativa.

Puedes usar las preguntas que quieras. La idea es que el resultado sea original, creativo, y hasta cómico, y si quieres agregar dificultad al desafío, te sugiero que intentes que el nombre de la cerveza se imprima entre comillas. Recuerda que hay diferentes formas de que la función print muestre las comillas sin cortar el string, y que ingrese la impresión final en al menos dos líneas utilizando saltos de línea dentro del código.



tipos de datos

En Python tenemos varios tipos o estructuras de datos, que son fundamentales en programación ya que almacenan información, y nos permiten manipularla.

texto (str)

```
"Python"  
"750"
```

números

```
int 250  
float 12.50
```

booleanos

```
True  
False
```

estructuras mutable ordenado duplicados

listas []



tuplas ()



sets {}

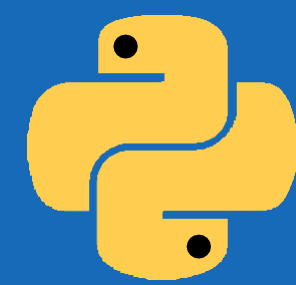


diccionarios {}



*: En Python 3.7+, existen consideraciones

** : key es única; value puede repetirse

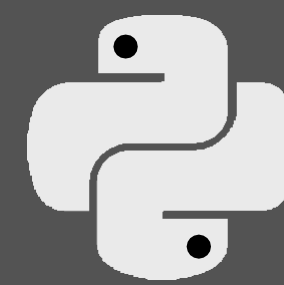


nombres de variables

Existen convenciones y buenas prácticas asociadas al nombre de las variables creadas en Python. Las mismas tienen la intención de facilitar la interpretabilidad y mantenimiento del código creado.

reglas

1. Legible: nombre de la variable es relevante según su contenido
2. Unidad: no existen espacios (puedes incorporar guiones bajos)
3. Hispanismos: omitir signos específicos del idioma español, como tildes o la letra ñ
4. Números: los nombres de las variables no deben empezar por números (aunque pueden contenerlos al final)
5. Signos/símbolos: no se deben incluir : " ' , < > / ? | \ () ! @ #
Σ % ^ & * ~ - +
6. Palabras clave: no utilizamos palabras reservadas por Python



variables

Las variables son espacios de memoria que almacenan valores o datos de distintos tipos, y (como su nombre indica) pueden variar. Se crean en el momento que se les asigna un valor, por lo cual en Python no requerimos declararlas previamente.

algunos ejemplos de uso

```
pais = "México"
```

```
nombre = input("Escribe tu nombre: ")  
print("Tu nombre es " + nombre)
```

```
num1 = 55  
num2 = 45  
print(num1 + num2)  
>> 100
```

integers & floats

Existen dos tipos de datos numéricos básicos en Python: int y float. Como toda variable en Python, su tipo queda definido al asignarle un valor a una variable. La función `type()` nos permite obtener el tipo de valor almacenado en una variable.

int

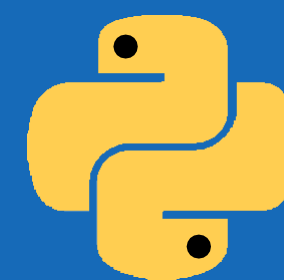
Int, o integer, es un número entero, positivo o negativo, sin decimales, de un largo indeterminado.

```
num1 = 7
print(type(num1))
>> <class 'int'>
```

float

Float, o "número de punto flotante" es un número que puede ser positivo o negativo, que a su vez contiene una o más posiciones decimales.

```
num2 = 7.525587
print(type(num2))
>> <class 'float'>
```

conversiones

Python realiza conversiones implícitas de tipos de datos automáticamente para operar con valores numéricos. En otros casos, necesitaremos generar una conversión de manera explícita.

```
int(var)  
>> <class 'int'>
```

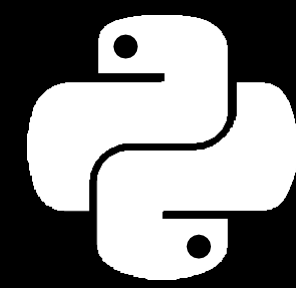


Convierte el dato en integer

```
float(var)  
>> <class 'float'>
```



Convierte el dato en float



formatear cadenas

Para facilitar la concatenación de variables y texto en Python, contamos con dos herramientas que nos evitan manipular las variables, para incorporarlas directamente al texto:

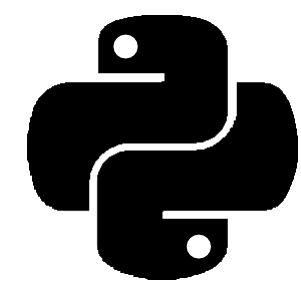
- Función format: se encierra las posiciones de las variables entre corchetes { }, y a continuación del string llamamos a las variables con la función format

```
print("Mi auto es {} y de matrícula  
{ }".format(color_auto,matricula))
```

- Cadenas literales (f-strings): a partir de Python 3.8, podemos anticipar la concatenación de variables anteponiendo **f** al string

```
print(f"Mi auto es {color_auto} y de  
matrícula {matricula}")
```

operadores matemáticos



Veamos cuáles son los operadores matemáticos básicos de Python, que utilizaremos para realizar cálculos:

Suma: +

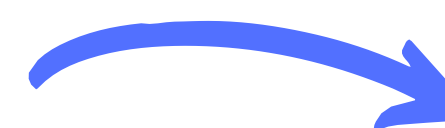
Resta: -

Multiplicación: *

División: /

Cociente (división "al piso"): //

Resto (módulo): %



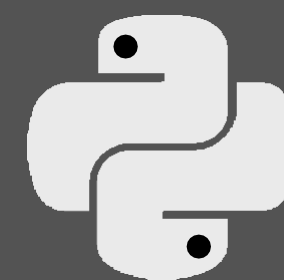
útil para detectar valores pares ;)

Potencia: **

Raíz cuadrada: **0.5



¡es un caso especial de potencia!



redondeo

El redondeo facilita la interpretación de los valores calculados al limitar la cantidad de decimales que se muestran en pantalla. También, nos permite aproximar valores decimales al entero más próximo.

`round(number, ndigits)`

valor a redondear ← cantidad de decimales
(si se omite, el resultado es entero)

algunos ejemplos de uso

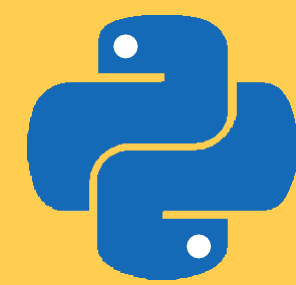
```
print(round(100/3))  
>> 33
```

```
print(round(12/7, 2))  
>> 1.71
```

Ejercicio 2

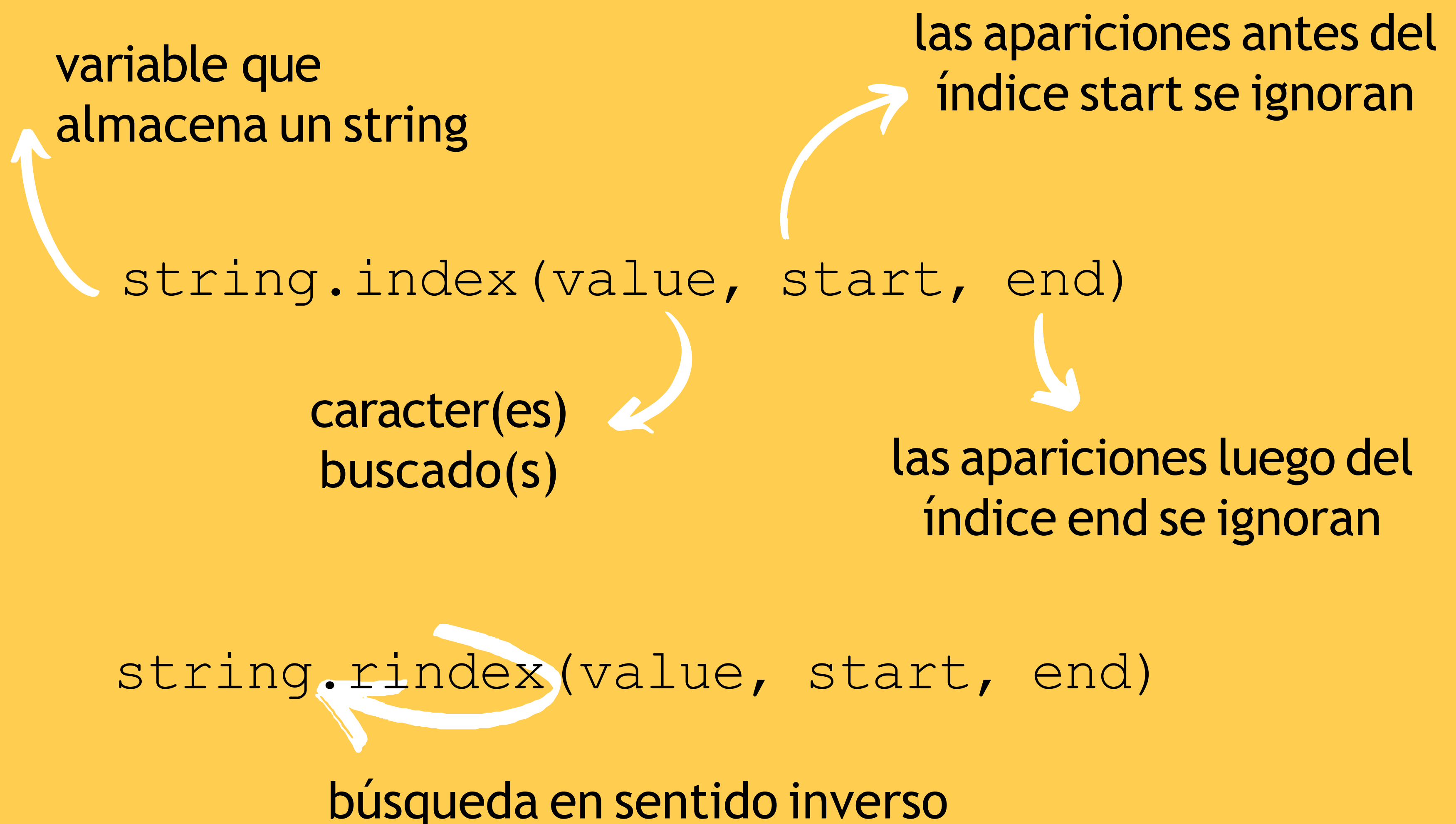
La situación es esta: tú trabajas en una empresa donde los vendedores reciben comisiones del 13% por sus ventas totales, y tu jefe quiere que ayudes a los vendedores a calcular sus comisiones creando un programa que les pregunte su nombre y cuánto han vendido en este mes. Tu programa le va a responder con una frase que incluya su nombre y el monto que le corresponde por las comisiones.

- Este programa debería comenzar preguntando cosas al usuario, por lo tanto, vas a necesitar input para poder recibir los ingresos del usuario y deberías usar variables para almacenar esos ingresos. Recuerda que los ingresos de usuarios se almacenan como strings. Por lo tanto, deberías convertir uno de esos ingresos en un float para poder hacer operaciones con él.
- ¿Y qué operaciones necesitas hacer? Bueno, calcular el 13% del número que haya ingresado el usuario. Es decir, que debes multiplicar ese número por 13 y luego dividirlo por 100. Recuerda almacenar ese resultado en una variable.
- Sería bueno que para imprimir en pantalla el resultado te asegures de que esa información no tenga más de dos decimales, para que sea fácil de leer, y luego organiza todo eso en un string al que debes dar formato.

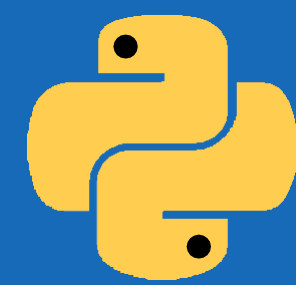


index()

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.



**: 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) paso

`string[start:stop:step]`

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

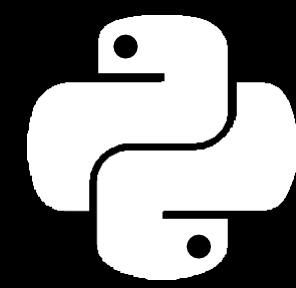
0 1 2 3 4 5 6 7 8 9

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

```
>> odnuM_aloH
```

H o l a _ M u n d o

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



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!')
```

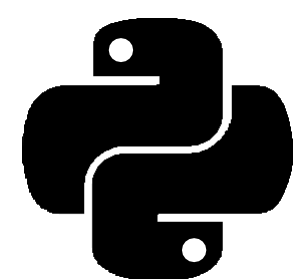
join() debe ser llamado desde una cadena que actúa como separador para unir dentro de una misma cadena resultante los elementos de una lista.

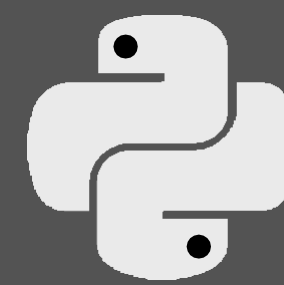
```
", ".join(["C", "C++", "Python", "Java"])  
>> 'C, C++, Python, Java'
```

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`).





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"
```

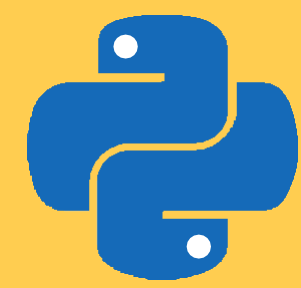
función `sort()`: ordena los elementos de la lista *en el lugar*

```
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`



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"
```

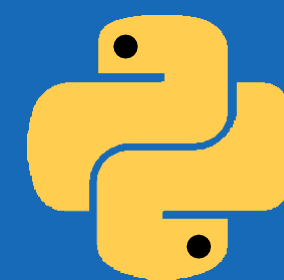
métodos para listar los nombres de las claves, valores, y pares clave:valor

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.*



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
```

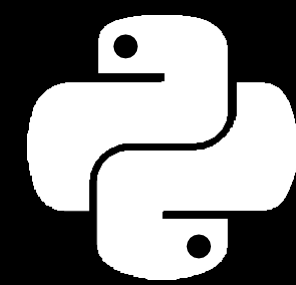
casting (conversión de tipos de datos)

```
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"]
```

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

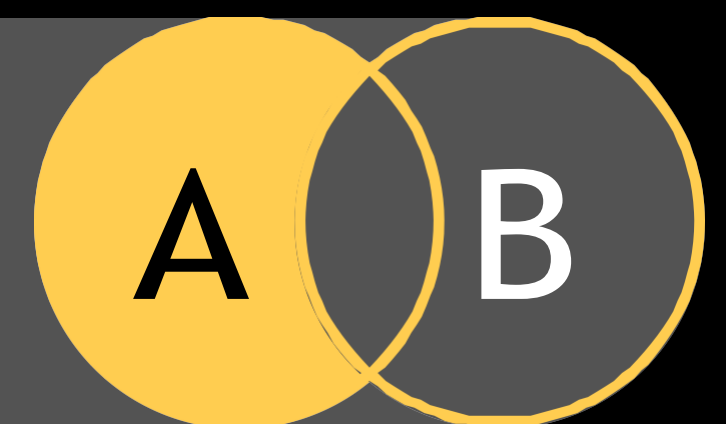
```
mi_set_c = mi_set_a.copy()
print(mi_set_c)
>> {1, 2, "tres"}
```

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'}
```

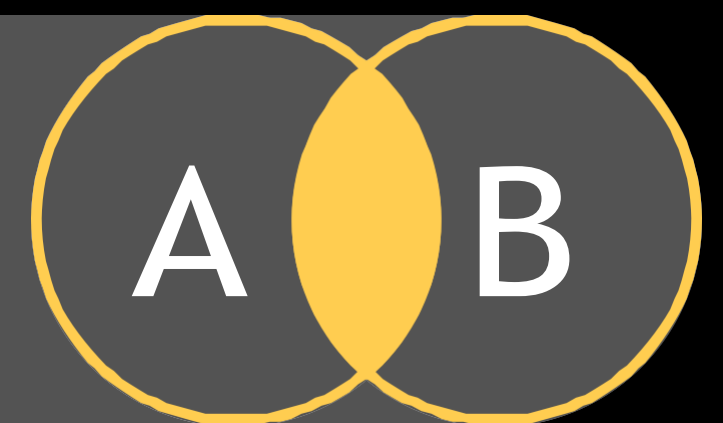


sets

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

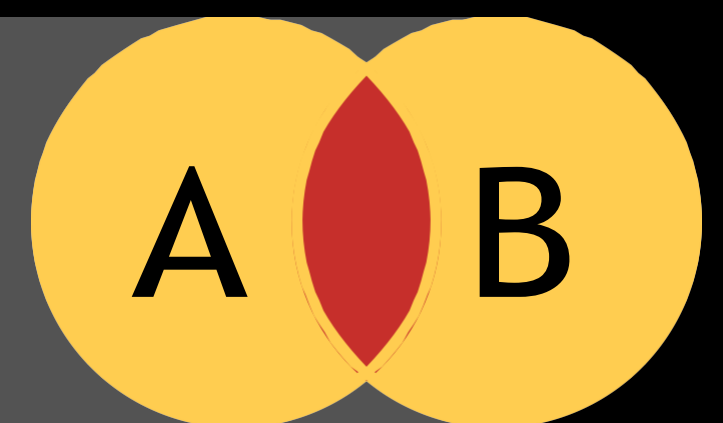
intersection_update(set) mantiene únicamente los elementos comunes a A y B

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



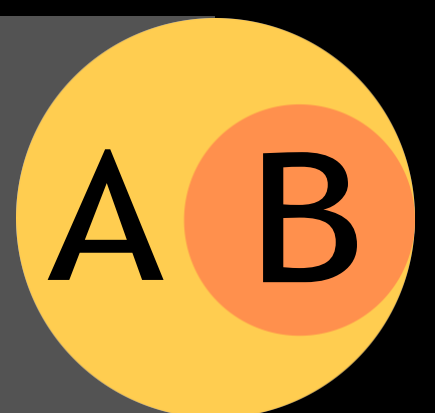
isdisjoint(set) devuelve True si A y B no tienen elementos en común

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



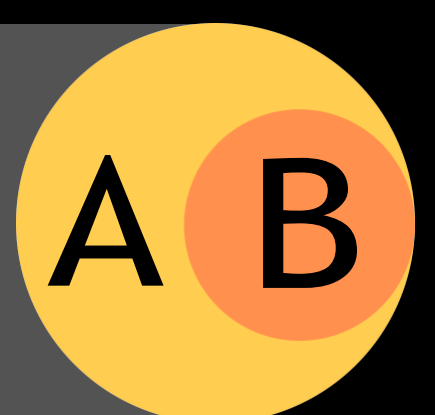
issubset(set) devuelve True si todos los elementos de B están presentes en A

```
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

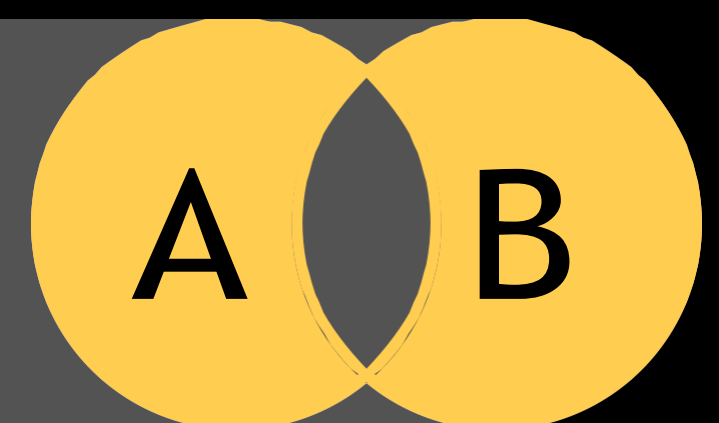
```
aleatorio = mi_set_a.pop()
print(aleatorio)
>> {2}
```

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

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

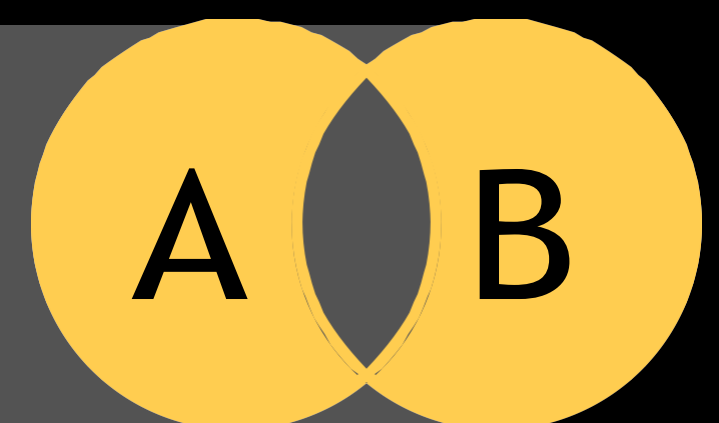
symmetric_difference(set) retorna todos los elementos de A y B, excepto aquellos que son comunes a los dos

```
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}
```

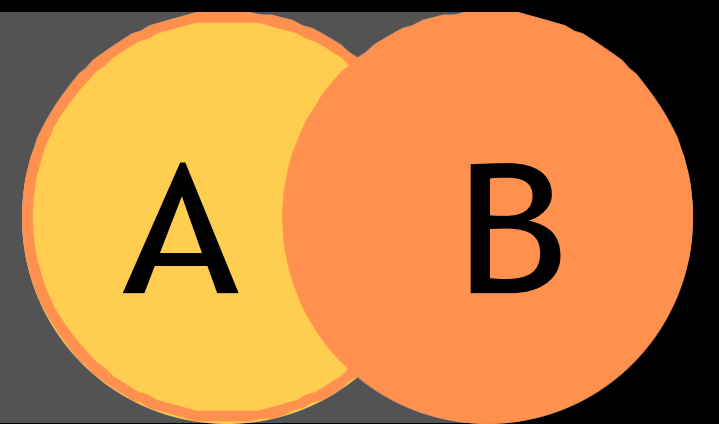


sets

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

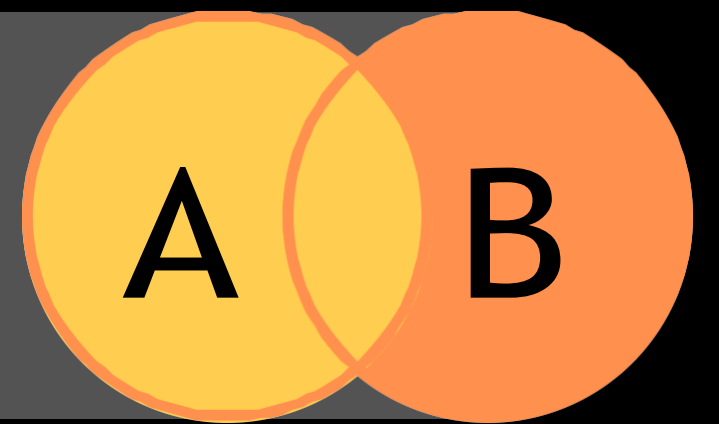
union(set) retorna un set resultado de combinar A y B (los datos duplicados se eliminan)

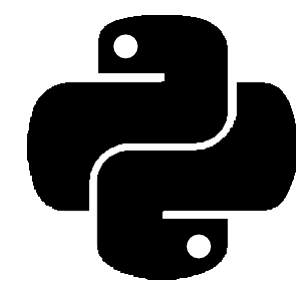
```
mi_set_c = mi_set_a.union(mi_set_b)
print(mi_set_c)
>> {1, 2, 3, 'tres'}
```



update(set) inserta en A los elementos de B

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





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**)

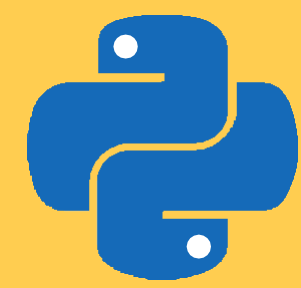
`not` no (invierte el valor del booleano)

Ejercicio 3

Ahora que ya sabes usar los métodos y las propiedades de los strings, que sabes indexar conjuntos de datos como los strings, las listas y los tuples, y sobre todo ahora que conoces todos los tipos de datos que necesitas para poder almacenar lo que sea, vas a poder encontrar una forma de programar un analizador de texto.

La consigna es la siguiente: vas a crear un programa que primero le pida al usuario que ingrese un texto. Puede ser un texto cualquiera: un artículo entero, un párrafo, una frase, un poema, lo que quiera. Luego, el programa le va a pedir al usuario que también ingrese tres letras a su elección y a partir de ese momento nuestro código va a procesar esa información para hacer cinco tipos de análisis y devolverle al usuario la siguiente información:

1. Primero: ¿cuántas veces aparece cada una de las letras que eligió? Para lograr esto, te recomiendo almacenar esas letras en una lista y luego usar algún método propio de string que nos permita contar cuantas veces aparece un sub string dentro del string. Algo que debes tener en cuenta es que al buscar las letras pueden haber mayúsculas y minúsculas y esto va a afectar el resultado. Lo que deberías hacer para asegurarte de que se encuentren absolutamente todas las letras es pasar, tanto el texto original como las letras a buscar, a minúsculas.
2. Segundo: le vas a decir al usuario cuántas palabras hay a lo largo de todo el texto. Y para lograr esta parte, recuerda que hay un método de string que permite transformarlo en una lista y que luego hay una función que permite averiguar el largo de una lista.
3. Tercero: nos va a informar cuál es la primera letra del texto y cuál es la última. Aquí claramente echaremos mano de la indexación.
4. Cuarto: el sistema nos va a mostrar cómo quedaría el texto si invirtiéramos el orden de las palabras. ¿Acaso hay algún método que permita invertir el orden de una lista, y otro que permita unir esos elementos con espacios intermedios? Piénsalo.
5. Y por último: el sistema nos va a decir si la palabra "Python" se encuentra dentro del texto. Esta parte puede ser algo complicada de imaginársela, pero te voy a dar una pista: puedes usar booleanos para hacer tu averiguación y un diccionario para encontrar la manera de expresarle al usuario tu respuesta.



operadores de comparación

Como su nombre lo indica, sirven para comparar dos o más valores. El resultado de esta comparación es un booleano (True/False)

==	igual a
!=	diferente a
>	mayor que
<	menor que
>=	mayor o igual que
<=	menor o igual que

Si la comparación resulta verdadera, devuelve el resultado True

Si dicha comparación es falsa, el resultado es False

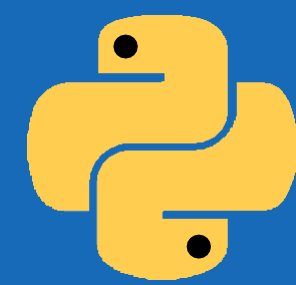
```
mi_bool = 5 >= 6  
print(mi_bool)  
>> False
```

```
mi_bool = 5 != 6  
print(mi_bool)  
>> True
```

```
mi_bool = 10 == 2*5  
print(mi_bool)  
>> True
```

```
mi_bool = 5 < 6  
print(mi_bool)  
>> True
```

operadores lógicos



Estos operadores permiten tomar decisiones basadas en múltiples condiciones.

```
a = 6 > 5
```

```
b = 30 == 15*3
```

and devuelve True si todas las condiciones son verdaderas

```
mi_bool = a and b  
print(mi_bool)  
>> False
```

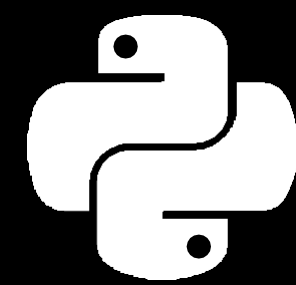
or devuelve True si al menos una condición es verdadera

```
mi_bool = a or b  
print(mi_bool)  
>> True
```

not devuelve True si el valor del booleano es False, y False si es True

```
mi_bool = not a  
print(mi_bool)  
>> False
```

control de flujo



El control de flujo determina el orden en que el código de un programa se va ejecutando. En Python, el flujo está controlado por **estructuras condicionales**, **loops** y **funciones**.

estructuras condicionales (if)

Expresión de resultado booleano

(True/False)

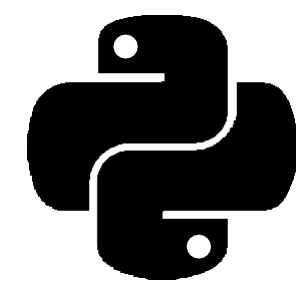
Los dos puntos (:) dan paso al código que se ejecuta si expresión = True

la indentación es obligatoria en Python

```
if expresión:
    código a ejecutarse
elif expresión:
    código a ejecutarse
elif expresión:
    código a ejecutarse
...
else:
    código a ejecutarse
```

else & elif son opcionales

pueden incluirse varias cláusulas elif



loops for

A diferencia de otros lenguajes de programación, los loops for en Python tienen la capacidad de iterar a lo largo de los elementos de cualquier secuencia (listas, strings, entre otros), en el orden en que dichos elementos aparecen.

Conceptos

- Loops/bucles: son secuencias de instrucciones de código que se ejecutan repetidas veces
- Iterables: son objetos en Python que se pueden recorrer (o iterar) a lo largo de sus elementos

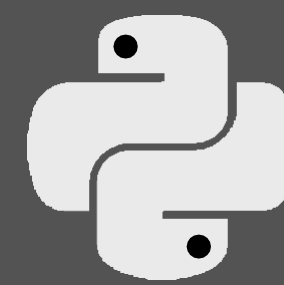
Cada uno de los elementos
que compone el iterable

Strings, listas, tuplas,
diccionarios, sets, rangos...



Código que se ejecuta con cada iteración

Podemos llamar al elemento que está siendo iterado o sus atributos a través del nombre definido en `item`, para utilizarlo en la expresión



loops while

Si bien los **loops while** son otro tipo de bucles, resultan más parecidos a los **condicionales if** que a los **loops for**. Podemos pensar a los loops while como una estructura condicional que se ejecuta en repetición, hasta que se convierte en falsa.

Estructura **condicional**

la **indentación**
es obligatoria en Python

```
while condición:  
    expresión  
else:  
    expresión
```

dos puntos (:)

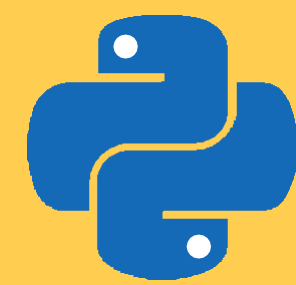
Este código se ejecutará cuando la condición se convierta en **False**

instrucciones especiales

Si el código llega a una instrucción **break**, se produce la **salida** del bucle.

La instrucción **continue** **interrumpe** la **iteración actual** dentro del bucle, llevando al programa a la parte superior del bucle.

La instrucción **pass** **no altera el programa**: ocupa un lugar donde se espera una declaración, pero no se desea realizar una acción.



range()

La función range() devuelve una secuencia de números dados 3 parámetros. Se utiliza fundamentalmente para controlar el número de ejecuciones de un loop o para crear rápidamente una serie de valores.

número a partir del cual
inicia el rango (incluido)

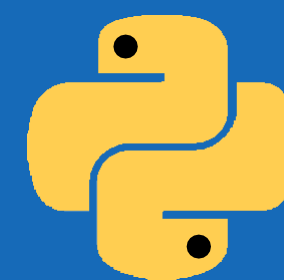
diferencia entre cada valor
consecutivo de la secuencia

```
range(valor_inicio, valor_final, paso)
```

número antes del cual el
rango finaliza (no incluido)

```
print(list(range(1, 13, 3)))  
>> [1, 4, 7, 10]
```

El único parámetro obligatorio es valor_final. Los valores predeterminados para valor_inicio y paso son 0 y 1 respectivamente.



enumerate()

La función `enumerate()` nos facilita llevar la cuenta de las iteraciones, a través de un contador de índices de un iterable, que se puede utilizar de manera directa en un loop, o convertirse en una lista de tuplas con el método `list()`.

Cualquier objeto que pueda ser iterado

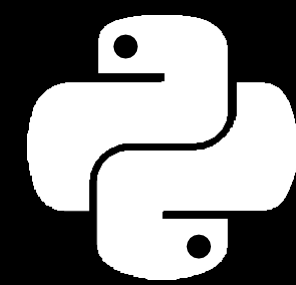
`enumerate(iterable, inicio)`

Valor [int] de inicio del índice (por defecto iniciado en 0)

```
print(list(enumerate("Hola")))
>> [(0, 'H'), (1, 'o'), (2, 'l'), (3, 'a')]
```

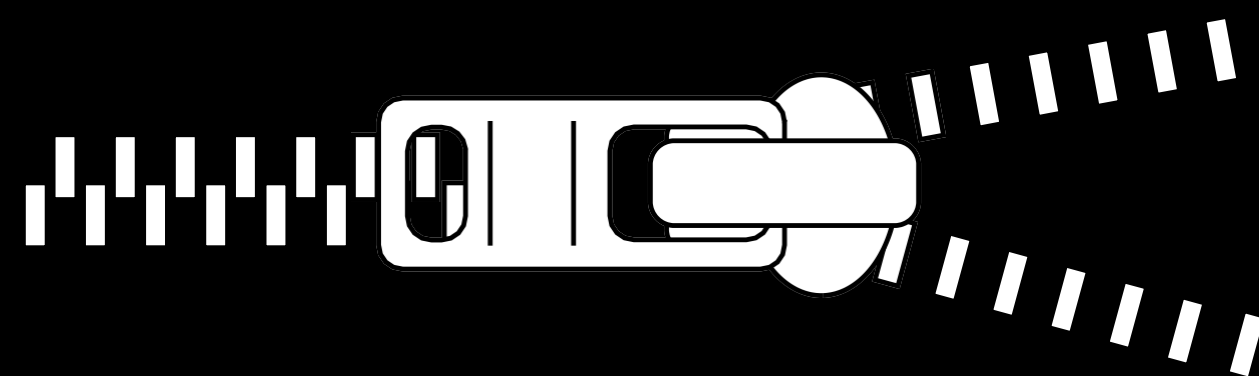
```
for indice, numero in enumerate([5.55, G, 7.50]):
    print(indice, numero)
```

```
>> 0 5.55
>> 1 G
>> 2 7.5
```



zip()

La función `zip()` crea un iterador formado por los elementos agrupados del mismo índice provenientes de dos o más iterables. Zip deriva de *zipper* (cremallera o cierre), de modo que es una analogía muy útil para recordar.

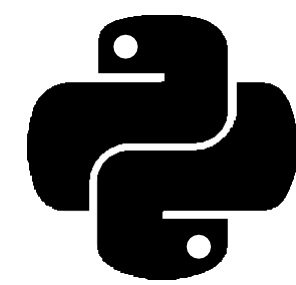


La función se detiene al cuando se agota el iterable con menor cantidad de elementos.

```
letras = ['w', 'x', 'c']
numeros = [50, 65, 90, 110, 135]
for letra, num in zip(letras, numeros):
    print(f'Letra: {letra}, y Número: {num}')
```

```
>> Letra: w, y Número: 50
>> Letra: x, y Número: 65
>> Letra: c, y Número: 90
```

min() & max()



La función min() retorna el item con el valor más bajo dentro de un iterable. La función max() funciona del mismo modo, devolviendo el valor más alto del iterable. Si el iterable contiene strings, la comparación se realiza alfabéticamente.

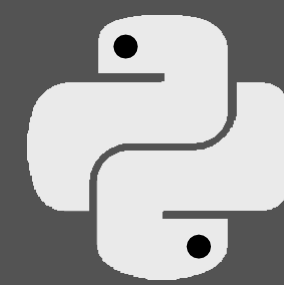
```
ciudades_habitantes = {"Tijuana":1810G45, "León":1579803}
```

```
lista_valores = [5**5, 12**2, 3050, 475*2]
```

```
print(min(ciudades_habitantes.keys()))  
>> León
```

```
print(max(ciudades_habitantes.values()))  
>> 1810G45
```

```
print(max(lista_valores))  
>> 3125
```

random

Python nos facilita un módulo (un conjunto de funciones disponibles para su uso) que nos permite generar selecciones pseudo-aleatorias* entre valores o secuencias.

Nombre del módulo

from random import *

* = Todos los métodos

También pueden importarse de manera independiente aquellos a utilizar.

randint(*min*, *max*): devuelve un **integer** entre dos valores dados (ambos **límites** incluidos)

uniform(*min*, *max*): devuelve un **float** entre un **valor mínimo y uno máximo**

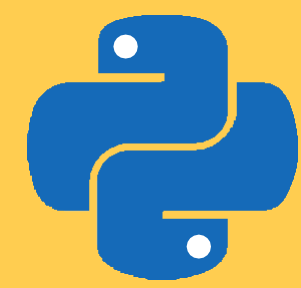
random(*sin parámetros*): devuelve un **float** entre 0 y 1

choice(*secuencia*): devuelve un **elemento al azar de una secuencia** de valores (listas, tuples, rangos, etc.)

shuffle(*secuencia*): toma una **secuencia** de valores mutable (como una lista), y la retorna **cambiando el orden de sus elementos aleatoriamente**.

** La mecánica en cómo se generan dichos valores aleatorios viene en realidad predefinida en "semillas". Si bien sirve para todos los usos habituales, no debe emplearse con fines de seguridad o criptográficos, ya que son vulnerables.*

comprensión de listas



La comprensión de listas ofrece una sintaxis más breve en la creación de una nueva lista basada en valores disponibles en otra secuencia. Vale la pena mencionar que la brevedad se logra a costo de una menor interpretabilidad.

cada elemento del iterable ↖
tuplas, sets, otras listas... ↗

```
nueva_lista= [expresion for item in iterable if condicion == True]
```

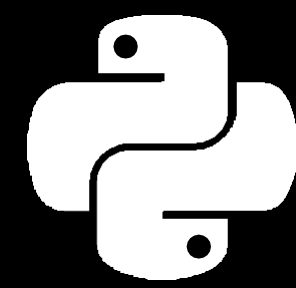
fórmula matemática *operación lógica*

Caso especial con else:

```
nueva_lista= [expresion if condicion == True else otra_expresion  
              for item in iterable]
```

Ejemplo:


```
nueva_lista = [num**2 for num in range(10) if num < 5]  
print(nueva_lista)  
>> [0, 1, 4, 9, 16]
```



match

En Python 3.10, se incorpora la coincidencia de patrones estructurales mediante las declaraciones **match** y **case**. Esto permite asociar acciones específicas basadas en las formas o patrones de tipos de datos complejos.

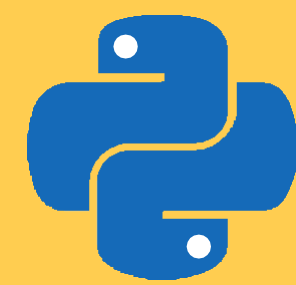
```
match objeto:
    case <patron_1>:
        <accion_1>
    case <patron_2>:
        <accion_2>
    case <patron_3>:
        <accion_3>
    case _:
        <accion_comodin>
```



El caracter `_` es un comodín que actúa como coincidencia si la misma no se produce en los casos anteriores.

Es posible detectar y deconstruir diferentes estructuras de datos: esto quiere decir que los patrones no son únicamente valores literales (strings o números), sino también estructuras de datos, sobre los cuales se buscan coincidencias de construcción.

funciones



Una función es un bloque de código que solamente se ejecuta cuando es llamada. Puede recibir información (en forma de *parámetros*), y devolver datos una vez procesados como resultado.

una función es definida
mediante la palabra clave def

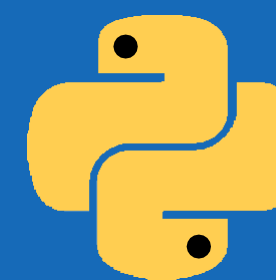


```
def mi_funcion(argumento):
```

Los argumentos contienen información que la función utiliza o transforma para devolver un resultado

```
mi_funcion(mi_argumento)
```

Para llamar a una función, basta utilizar su nombre, entregando los argumentos que la misma requiere entre paréntesis.



return

Para que una función pueda devolver un valor (y que pueda ser almacenado en una variable, por ejemplo), utilizamos la declaración return.

```
def mi_funcion():  
    return [algo]
```



La declaración return provoca la salida de la función
Cualquier código que se encuentre después en el bloque de la función, no se ejecutará

```
resultado = mi_funcion()
```



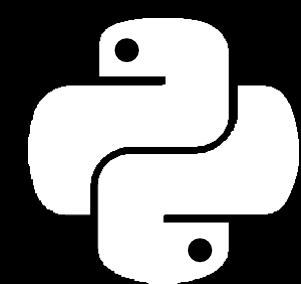
La variable resultado almacenará el valor devuelto por la función mi_funcion()

funciones dinámicas

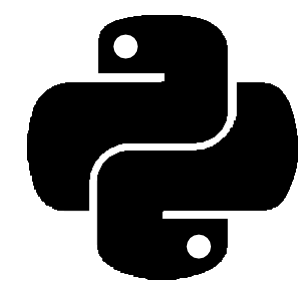
La integración de diferentes herramientas de control de flujo, nos permite crear funciones dinámicas y flexibles. Si debemos utilizarlas varias veces, lograremos un programa más limpio y sencillo de mantener, evitando repeticiones de código.

- Funciones
- Loops (for/while)
- Estructuras condicionales
- Palabras clave (return, break, continue, pass)

```
def mi_funcion(argumento):  
    for item in ...  
        if a == b ...  
            ...  
        else:  
            return ...  
    return ...
```



interacción entre funciones



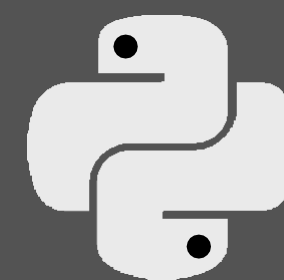
Las salidas de una determinada función pueden convertirse en entradas de otras funciones. De esa manera, cada función realiza una tarea definida, y el programa se construye a partir de la interacción entre funciones.

```
def funcion_1() :  
    ...  
    return a
```

```
def funcion_2(a) :  
    ...  
    return b
```

```
def funcion_3(b) :  
    ...  
    return c
```

```
def funcion_4(a, c) :  
    ...  
    return d
```



*args

En aquellos casos en los que no se conoce de antemano el número exacto de argumentos que se deben pasar a una función, se debe utilizar la sintaxis `*args` para referirse a todos los argumentos adicionales luego de los obligatorios.

*El nombre `*args` no es mandatorio pero si es sugerido como buena práctica. Cualquier nombre iniciado en `*` referirá a estos argumentos de cantidad variable.*

La función recibirá los argumentos indefinidos `*args` en forma de tupla, a los que se podrá acceder o iterar de las formas habituales dentro del bloque de código de la función.

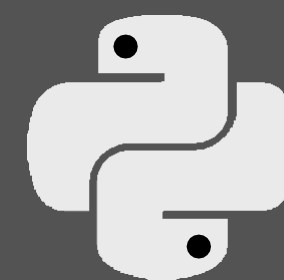
```
def mi_funcion(arg_1, arg_2, *args):
```

```
mi_funcion("ejemplo", 25, 40, 75, 10):
```

arg_1

arg_2

args = (40, 75, 10)



*args

En aquellos casos en los que no se conoce de antemano el número exacto de argumentos que se deben pasar a una función, se debe utilizar la sintaxis `*args` para referirse a todos los argumentos adicionales luego de los obligatorios.

*El nombre `*args` no es mandatorio pero si es sugerido como buena práctica. Cualquier nombre iniciado en `*` referirá a estos argumentos de cantidad variable.*

La función recibirá los argumentos indefinidos `*args` en forma de tupla, a los que se podrá acceder o iterar de las formas habituales dentro del bloque de código de la función.

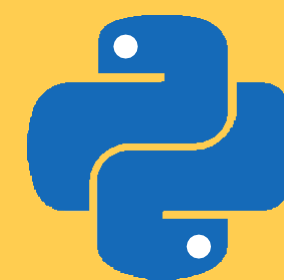
```
def mi_funcion(arg_1, arg_2, *args):
```

```
mi_funcion("ejemplo", 25, 40, 75, 10):
```

arg_1

arg_2

args = (40, 75, 10)



****kwargs**

Los argumentos con palabras clave (*keyword arguments*, abreviado *kwargs*), sirven para identificar el argumento por su nombre, independientemente de la posición en la que se pasen a su función. Si no se conoce de antemano su cantidad, se utiliza el parámetro ****kwargs** que los agrupa en un diccionario.

*Al igual que para *args, el nombre **kwargs no es mandatorio pero si es sugerido como buena práctica. Cualquier nombre iniciado en ** referirá a estos argumentos de cantidad variable.*

```
def atributos_persona (**kwargs) :
```

```
    atributos_persona(ojos="azules", pelo="corto")
```



```
kwargs = {'ojos': 'azules', 'pelo': 'rubio'}
```


Ejercicio 5

El programa va a elegir una palabra secreta y le va a mostrar al jugador solamente una serie de guiones que representa la cantidad de letras que tiene la palabra. El jugador en cada turno deberá elegir una letra y si la letra se encuentra en la palabra oculta, el sistema le va a mostrar en qué lugares se encuentra. Pero si el jugador dice una letra que no se encuentra en la palabra oculta, pierde una vida.

En el juego real del ahorcado, cada vez que perdemos una vida, se va completando el dibujo del ahorcado miembro por miembro. Pero en nuestro caso, como aún no tenemos elementos gráficos, simplemente le vamos a decir que tiene seis vidas y se las iremos descontando de una en una, cada vez que el jugador elija una letra incorrecta.

Si se agotan las vidas antes de adivinar la palabra, el jugador pierde. Pero si adivina la palabra completa antes de perder todas las vidas, el jugador gana.

Parece sencillo, pero ¿cómo diseñamos todo este código? Bueno, aquí van algunas ayudas:

- Primero vas a crear un código que importe el método choice, ya que lo vas a necesitar para que el sistema pueda elegir una palabra al azar dentro de una lista de palabras que también vas a crear al comienzo.
- Luego de eso, vas a crear tantas funciones como creas necesarias para que el programa haga cosas como pedirle al usuario que elija una letra, para corroborar si lo que el usuario ha ingresado es una letra válida, para chequear si la letra ingresada se encuentra en la palabra o no, para verificar si ha ganado o no, etc.
- Recuerda escribir primero las funciones y luego el código que las implementa ordenadamente.