# 3.0.-Est.Datos: Cadenas

## 3.0 Cadenas

Las cadenas son una secuencia de caracteres. En Python, las cadenas son objetos y tienen métodos asociados. En esta unidad vamos a ver cómo trabajar con cadenas en Python.

### 1. Acceso a los caracteres de una cadena

En Python, las cadenas son secuencias de caracteres. Puedes acceder a los caracteres de una cadena utilizando el operador de indexación `[]`. Por ejemplo, si tienes una cadena `fruta` y quieres acceder al segundo carácter de la cadena, puedes hacerlo de la siguiente manera:

```Python
>>> fruta = 'banana'
>>> letra = fruta[1]
```

La segunda sentencia extrae el carácter en la posición del índice 1 de la variable `fruta` y la asigna a la variable `letra`.

La expresión que hay dentro de los corchetes es llamada *índice*. El índice indica qué carácter de la secuencia quieres (de ahí el nombre).

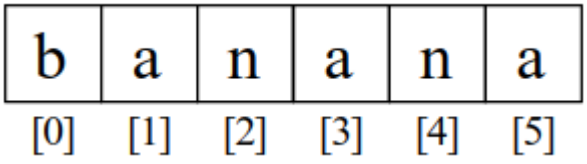
Pero podrías no obtener lo que esperas:

```
>>> print(letra)
a
```

Para la mayoría de las personas, la primera letra de “banana” es “b”, no “a”. Pero en Python, el índice es un desfase desde el inicio de la cadena, y el desfase de la primera letra es cero.

```
>>> letra = fruta[0]
>>> print(letra)
b
```

Así que “b” es la letra 0 (“cero”) de “banana”, “a” es la letra con índice 1, y “n” es la que tiene índice 2, etc.




Representación de una Cadena de texto

### 1.1. Índices de Cadenas

Puedes usar cualquier expresión, incluyendo variables y operadores, como un índice, pero el valor del índice tiene que ser un entero. De otro modo obtendrás:

```
>>> letra = fruta[1.5]
TypeError: string indices must be integers
```

### 1.2. Rebanado (slicing) de una cadena

Un segmento de una cadena es llamado *rebanado*. Seleccionar un rebanado es similar a seleccionar un carácter:

```
>>> s = 'Monty Python'
>>> print(s[0:5])
Monty
>>> print(s[6:12])
Python
```

El operador `[n:m]` retorna la parte de la cadena desde el "n-ésimo" carácter hasta el "m-ésimo" carácter, **incluyendo el primero, pero excluyendo el último**.

Si omites el primer índice (antes de los dos puntos), el rebanado comienza desde el inicio de la cadena. Si omites el segundo índice, el rebanado va hasta el final de la cadena:

```
>>> fruta = 'banana'
>>> fruta[:3]
'ban'
>>> fruta[3:]
'ana'
```

Si el primer índice es mayor que o igual que el segundo, el resultado es una *cadena vacía*, representado por dos comillas:

```
>>> fruta = 'banana'
>>> fruta[3:3]
''
```

Una cadena vacía no contiene caracteres y tiene un tamaño de 0, pero fuera de esto es lo mismo que cualquier otra cadena.

### 1.3. Las cadenas son inmutables

Puede ser tentador utilizar el operador [] en el lado izquierdo de una asignación, con la intención de cambiar un carácter en una cadena. Por ejemplo:

```
>>> saludo = 'Hola, mundo!'
>>> saludo[0] = 'J'
TypeError: 'str' object does not support item assignment
```

El “objeto” en este caso es la cadena y el “ítem” es el carácter que tratamos de asignar. Por ahora, un *objeto* es la misma cosa que un valor, pero vamos a redefinir esa definición después. Un *ítem* es uno de los valores en una secuencia.

La razón por la cual ocurre el error es que las cadenas son *inmutables*, lo cual significa que no puedes modificar una cadena existente. Lo mejor que puedes hacer es crear una nueva cadena que sea una variación de la original:

```
>>> saludo = 'Hola, mundo!'
>>> nuevo_saludo = 'J' + saludo[1:]
>>> print(nuevo_saludo)
Jola, mundo!
```

Este ejemplo concatena una nueva letra a una parte de `saludo`. Esto no tiene efecto sobre la cadena original.

## 2. Operando con cadenas

Las cadenas en Python soportan operadores como `+`, `*`, `==` e `in`. Métodos como `len` para obtener la longitud , métodos

### 2.1. Multiplicando `*` y concatenando de cadenas `+`

En general, no puedes realizar operaciones matemáticas con cadenas, incluso si los caracteres parecen números. El siguiente código es incorrecto:

```
>>> fruta = 'banana'
>>> 'n' + 1
TypeError: Can't convert 'int' object to str implicitly
```

El error que obtienes es un `TypeError`, que significa que estás intentando operar con tipos incompatibles. En este caso, estás intentando sumar un carácter a un número.

Pero puedes realizar otras operaciones con cadenas, como la concatenación:

```
>>> fruta = 'banana'
>>> prefijo = 'an'
>>> prefijo + fruta
'ananana'
```

También puedes multiplicar una cadena por un número entero:

```
>>> prefijo = 'an'
>>> prefijo * 3
'ananan'
```

### 2.2. Obtener el tamaño de una cadena usando `len`

`len` es una función nativa que devuelve el número de caracteres en una cadena:

```
>>> fruta = 'banana'
>>> len(fruta)
6
```

Para obtener la última letra de una cadena, podrías estar tentado a probar algo como esto:

```
>>> tamaño = len(fruta)
>>> ultima = fruta[tamaño]
IndexError: string index out of range
```

La razón de que haya un `IndexError` es que ahí no hay ninguna letra en "banana" con el índice 6. Puesto que empezamos a contar desde cero, las seis letras están enumeradas desde 0 hasta 5. Para obtener el último carácter, tienes que restar 1 a `length`:

```
>>> ultima = fruta[tamaño-1]
>>> print(ultima)
a
```

Alternativamente, puedes usar índices negativos, los cuales cuentan hacia atrás desde el final de la cadena. La expresión `fruta[-1]` devuelve la última letra, `fruta[-2]` la penúltima letra, y así sucesivamente.

**IMPORTANTE**: En las secuencias, el índice empieza en la posición `0` y termina en la posición `len() - 1`

### 2.3. El operador `in`

La palabra `in` es un operador booleano que toma dos cadenas y regresa `True` si la primera cadena aparece como una subcadena de la segunda:

```
>>> 'a' in 'banana'
True
>>> 'semilla' in 'banana'
False
```

### 2.4. Comparación de cadenas ==, >, <...

Los operadores de comparación funcionan en cadenas. Para ver si dos cadenas son iguales:

```
if palabra == 'banana':
    print('Muy bien, bananas.')
```

Otras operaciones de comparación son útiles para poner palabras en orden alfabético:

```
if palabra < 'banana':
    print('Tu palabra, ' + palabra + ', está antes de banana.')
elif palabra > 'banana':
    print('Tu palabra, ' + palabra + ', está después de banana.')
else:
    print('Muy bien, bananas.')
```

Python no maneja letras mayúsculas y minúsculas de la misma forma que la gente lo hace. Todas **las letras mayúsculas van antes que todas las letras minúsculas**, por ejemplo:

```
Tu palabra, Piña, está antes que banana.
```

Una forma común de manejar este problema es convertir cadenas a un formato estándar, como todas a minúsculas, antes de llevar a cabo la comparación.

### 2.5. El operador de formato `%`

El *operador de formato* `%` nos permite construir cadenas, reemplazando partes de las cadenas con datos almacenados en variables. Cuando lo aplicamos a enteros, `%` es el operador módulo. Pero cuando es aplicado a una cadena, `%` es el operador de formato.

El primer operando es la *cadena a formatear*, la cual contiene una o más *secuencias de formato* que especifican cómo el segundo operando es formateado. El resultado es una cadena.

Por ejemplo, la secuencia de formato `%d` significa que el segundo operando debería ser formateado como un entero (“d” significa “decimal”):

```
>>> camellos = 42
>>> '%d' % camellos
'42'
```

El resultado es la cadena ‘42’, el cual no debe ser confundido con el valor entero 42.

Una secuencia de formato puede aparecer en cualquier lugar en la cadena, así que puedes meter un valor en una frase:

```
>>> camellos = 42
>>> 'Yo he visto %d camellos.' % camellos
'Yo he visto 42 camellos.'
```

Si hay más de una secuencia de formato en la cadena, el segundo argumento tiene que ser una tupla. Cada secuencia de formato es relacionada con un elemento de la tupla, en orden.

El siguiente ejemplo usa `%d` para formatear un entero, `%g` para formatear un número de punto flotante (no preguntes por qué), y `%s` para formatear una cadena:

```
>>> 'En %d años yo he visto %g %s.' % (3, 0.1, 'camellos')
'En 3 años yo he visto 0.1 camellos.'
```

El número de elementos en la tupla debe coincidir con el número de secuencias de formato en la cadena. El tipo de los elementos también debe coincidir con la secuencia de formato:

```
>>> '%d %d %d' % (1, 2)
TypeError: not enough arguments for format string
>>> '%d' % 'dolares'
TypeError: %d format: a number is required, not str
```

En el primer ejemplo, no hay suficientes elementos; en el segundo, el elemento es de un tipo incorrecto.

El operador de formato es poderoso, pero puede ser difícil de usar. Puedes leer más al respecto en

https://docs.python.org/library/stdtypes.html#printf-style-string-formatting.

## 3. Iteradores: Recorriendo una cadena mediante un bucle

Muchos de los cálculos requieren procesar una cadena carácter por carácter. Frecuentemente empiezan desde el inicio, seleccionando cada carácter presente, haciendo algo con él, y continuando hasta el final. Este patrón de procesamiento es llamado un *iterador*. Una manera de escribir un iterador es con un bucle `while`:

```
indice = 0
while indice < len(fruta):
    letra = fruta[indice]
    print(letra)
    indice = indice + 1
```

Este bucle recorre la cadena e imprime cada letra en una línea cada una. La condición del bucle es `indice < len(fruta)`, así que cuando `indice` es igual al tamaño de la cadena, la condición es falsa, y el código del bucle no se ejecuta. El último carácter accedido es el que tiene el índice `len(fruta)-1`, el cual es el último carácter en la cadena.

Otra forma de escribir un iterador es con un bucle `for`:

```
for caracter in fruta:
    print(caracter)
```

Cada vez que iteramos el bucle, el siguiente carácter en la cadena es asignado a la variable `caracter`. El ciclo continúa hasta que no quedan caracteres.

### 3.1. Iterando y contando

El siguiente programa cuenta el número de veces que la letra “a” aparece en una cadena:

```
palabra = 'banana'
contador = 0
for letra in palabra:
    if letra == 'a':
        contador = contador + 1
print(contador)
```

Este programa demuestra otro patrón de computación llamado *contador*. La variable `contador` es inicializada a 0 y después se incrementa cada vez que una “a” es encontrada. Cuando el bucle termina, `contador` contiene el resultado: el número total de a’s.

## 5. Métodos de cadenas

Las cadenas son un ejemplo de *objetos* en Python. Un objeto contiene tanto datos (el valor de la cadena misma) como *métodos*, los cuales son efectivamente funciones que

están implementadas dentro del objeto y que están disponibles para cualquier *instancia* del objeto.

Python tiene una función llamada `dir` la cual lista los métodos disponibles para un objeto. La función `type` muestra el tipo de un objeto y la función `dir` muestra los métodos disponibles.

```
>>> cosa = 'Hola mundo'
>>> type(cosa)
<class 'str'>
>>> dir(cosa)
['capitalize', 'casefold', 'center', 'count', 'encode',
'endswith', 'expandtabs', 'find', 'format', 'format_map',
'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind',
'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
>>> help(str.capitalize)
Help on method_descriptor:

capitalize(...)
    S.capitalize() -> str

    Return a capitalized version of S, i.e. make the first character
have upper case and the rest lower case.
>>>
```

Llamar a un *método* es similar a llamar una función (esta toma argumentos y devuelve un valor) pero la sintaxis es diferente. Llamamos a un método uniendo el nombre del método al de la variable, usando un punto como delimitador. Aunque la función `dir` lista los métodos y puedes usar la función `help` para obtener una breve documentación de un método, una mejor fuente de documentación para los métodos de cadenas se puede encontrar en Métodos en inglés y Métodos en castellano

Por ejemplo, el método `upper` toma una cadena y devuelve una nueva cadena con todas las letras en mayúscula:

En vez de la sintaxis de función `upper(palabra)`, éste utiliza la sintaxis de método `palabra.upper()`.

```
>>> palabra = 'banana'
>>> nueva_palabra = palabra.upper()
>>> print(nueva_palabra)
BANANA
```

Esta forma de notación con punto especifica el nombre del método, `upper`, y el nombre de la cadena al que se le aplicará el método, `palabra`. Los paréntesis vacíos indican que el método no toma argumentos.

Una llamada a un método es conocida como una *invocación*; en este caso, diríamos que estamos invocando `upper` en `palabra`.

Por ejemplo, existe un método de cadena llamado `find` que busca la posición de una cadena dentro de otra:

```
>>> palabra = 'banana'
>>> indice = palabra.find('a')
>>> print(indice)
1
```

En este ejemplo, invocamos `find` en `palabra` y pasamos la letra que estamos buscando como un parámetro.

El método `find` puede encontrar subcadenas así como caracteres:

```
>>> palabra.find('na')
2
```

También puede tomar como un segundo argumento el índice desde donde debe empezar:

```
>>> palabra.find('na', 3)
4
```

Una tarea común es eliminar los espacios en blanco (espacios, tabs, o nuevas líneas) en el inicio y el final de una cadena usando el método `strip`:

```
>>> linea = '  Aquí vamos  '
>>> linea.strip()
'Aquí vamos'
```

Algunos métodos como `startswith` devuelven valores booleanos.

```
>>> linea = 'Que tengas un buen día'
>>> linea.startswith('Que')
True
>>> linea.startswith('q')
False
```

Puedes notar que `startswith` requiere que el formato (mayúsculas y minúsculas) coincida, de modo que a veces tendremos que tomar la línea y cambiarla completamente a minúsculas antes de hacer la verificación, utilizando el método `lower`.

```
>>> linea = 'Que tengas un buen día'
>>> linea.startswith('q')
False
>>> linea.lower()
'que tengas un buen día'
>>> linea.lower().startswith('q')
True
```

En el último ejemplo, el método `lower` es llamado y después usamos `startswith` para ver si la cadena resultante en minúsculas comienza con la letra “q”. Siempre y cuando seamos cuidadosos con el orden, podemos hacer múltiples llamadas a métodos en una sola expresión.

Es recomendable tener la ayuda cerca y consultar los métodos disponibles y su funcionamiento, siempre actualizado a las últimas versiones. No obstante, aquí hay una lista no exhaustiva de métodos:

### 5.1. string.capitalize()

El método capitalize() devuelve una copia de la cadena con su primera letra en mayúscula. Ejemplo:

```
>>> texto = "mi diario python"
>>> texto.capitalize()
'Mi diario python'
```

### 5.2. string.endswith(sufijo)

El método endswith() devuelve True si la cadena termina con el sufijo especificado. Ejemplo:

```
>>> texto = "mi diario python"
>>> texto.endswith("python")
True
>>> texto.endswith("thon")
True
>>> texto.endswith("py")
False
```

### 5.3. string.expandtabs(tamaño_de_tab=8)

El método expandtabs devuelva una copia de la cadena en la que todos los caracteres de las pestañas se reemplazan por uno o más espacios, según la columna actual y el tamaño de la pestaña. Para expandir la cadena, la columna actual se establece en cero y la cadena se examina carácter por carácter. Si el carácter es una pestaña ( `t`), se insertan uno o más caracteres de espacio en el resultado hasta que la columna actual sea igual a la siguiente posición de la pestaña. Ejemplo:

```
>>> texto = "mitdiariotpython"
>>> texto.expandtabs(4)
'mi  diario  python'
```

### string.find(sub)

Devuelve el índice más bajo de la cadena en la subcadena *sub* se encuentra dentro de la rebanada `s[start:end]`. Devuelve -1 si no se encuentra el sub. Ejemplo:

```
>>> texto = "mi diario python"
>>> texto.find("mi")
0
>>> texto.find("m")
0
>>> texto.find("i")
1
>>> texto.find("python")
10
>>> texto.find("py")
```

```
10
>>> texto.find("p y")
-1
```

### 5.4. string.format()

Nos permite realizar una operación de formato de cadena. La cadena en la que se llama a este método puede contener texto literal o campos de reemplazo delimitados por llaves `{}`. Cada campo de reemplazo contiene el índice numérico de un argumento posicional o el nombre de un argumento de palabra clave. Devuelve una copia de la cadena donde cada campo de reemplazo se reemplaza con el valor de la cadena del argumento correspondiente. Ejemplo:

```
>>> "La suma de 1 + 2 es {0}".format(1+2)
'La suma de 1 + 2 es 3'
```

### 5.6. string.index(sub)

El método index es muy similar al método find. Con la diferencia de que cuando no se encuentra la subcadena, index lanza un ValueError.

```
>>> texto = "mi diario python"
>>> texto.index("mi")
0
>>> texto.index("PYTHON")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>>
```

### 5.7. string.isalpha()

Devuelve verdadero si todos los caracteres de la cadena son alfanuméricos y hay al menos un carácter, de lo contrario es falso. Ejemplo:

```
>>> texto = "mi diario python"
>>> texto.isalpha()
False
>>> "midiariopython".isalpha()
True
```

Si te preguntas ¿por qué “mi diario python” ha lanzado False? Es porque los espacios no son un carácter alfanumérico.

### 5.8. string.isdigit()

El método isdigit() devuelve True si todos los caracteres de la cadena son dígitos. Ejemplo:

```
>>> texto = "mi diario python"
>>> digitos = "12345"
>>> texto.isdigit()
False
>>> digitos.isdigit()
```

```
True
```

### 5.9. string.isspace()

El método isspace() devuelve True si solo hay caracteres de espacio en blanco. Ejemplo:

```
>>> " ".isspace()
True
>>> "  a".isspace()
False
```

### string.lower()

El método lower devuelve una copia de la cadena con todos sus caracteres en minúsculas. Ejemplo:

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

### 5.10. string.upper()

El método upper() devuelve la una copia de la cadena con todos su caracteres en mayúsculas. Ejemplo:

```
>>> texto = "mi diario python"
>>> texto.upper()
'MI DIARIO PYTHON'
```

### 5.11. string.lstrip(chars)

El método lstrip devuelve una copia de la cadena con los caracteres iniciales eliminados. El argumento *chars* es una cadena que especifica el conjunto de caracteres que se eliminarán. Ejemplo:

```
>>> web = "www.pythondiario.com"
>>> web.lstrip("w.")
'pythondiario.com'
>>>
```

### 5.12. string.replace(string_viejo, string_nuevo)

El método replace() devuelve una copia de la cadena con la subcadena vieja remplazada por una nueva. Veamos un ejemplo para entenderlo mejor:

```
>>> cadena = "Hola Mundo"
>>> cadena.replace("Mundo", "Internet")
'Hola Internet'
```

### 5.13. string.partition(char)

El método partition() divide la cadena en la primera aparición de char y devuelve una tupla que contiene la parte anterior a char, el mismo char, y la parte posterior de char. Suena un poco confuso, veamos un ejemplo:

```
>>> cadena = "Hola Mundo"
>>> cadena.partition("la")
('Ho', 'la', ' Internet')
```

### 5.14. string.title()

El método title() devuelve una copia de la cadena donde las palabras comienzan con una letra mayúscula. Veamos un ejemplo:

```
>>> cadena = "mi diario python"
>>> cadena.title()
'Mi Diario Python'
```

### 5.15. string.swapcase()

El método swapcase() devuelve una copia de la cadena con los caracteres en mayúsculas convertidos en minúsculas y viceversa.

Ejemplo:

```
>>> cadena = "Mi Diario Python"
>>> cadena.swapcase()
'mI dIARIO pYTHON'
```

### 5.16. string.startswith(prefijo)

El método startswith() devuelve True si la cadena comienza con el prefijo, de lo contrario devuelve False. Ejemplo:

```
>>> cadena = "Mi Diario Python"
>>> cadena.startswith("Mi")
True'
```

### 5.17. string.split(sep)

El método split() devuelve una lista de las palabras en la cadena, utilizando a sep como la cadena delimitadora. Ejemplo:

```
>>> cadena = "Luis,Jose,Maria,Sofia,Miguel"
>>> cadena.split(",")
['Luis', 'Jose', 'Maria', 'Sofia', 'Miguel']
```

### 5.18. string.zfill(ancho)

El método zfill() devuelve una copia de la cadena que se rellena con 0 dígitos ASCII para hacer una cadena de *ancho* de longitud . Ejemplo:

```
>>> "356".zfill(6)
'000356'
```

## 6. Analizando cadenas

Frecuentemente, queremos examinar una cadena para encontrar una subcadena. Por ejemplo, si se nos presentaran una seria de líneas con el siguiente formato:

```
From stephen.marquard@uct.ac.za Sat Jan 5 09:14:16 2008
```

y quisiéramos obtener únicamente la segunda parte de la dirección de correo (esto es, `uct.ac.za`) de cada línea, podemos hacer esto utilizando el método `find` y una parte de la cadena.

Primero tenemos que encontrar la posición de la arroba en la cadena. Después, tenemos que encontrar la posición del primer espacio *después* de la arroba. Y después partiremos la cadena para extraer la porción de la cadena que estamos buscando.

```
>>> dato = 'From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008'
>>> arrobapos = dato.find('@')
>>> print(arrobapos)
21
>>> espos = dato.find(' ',arrobapos)
>>> print(espos)
31
>>> direccion = dato[arrobapos+1:espos]
>>> print(direccion)
uct.ac.za
>>>
```

Utilizamos una versión del método `find` que nos permite especificar la posición en la cadena desde donde queremos que `find` comience a buscar. Cuando recortamos una parte de una cadena, extraemos los caracteres desde “uno después de la arroba hasta, *pero no incluyendo* , el carácter de espacio”.

La documentación del método `find` está disponible en

Métodos en castellano.

## 7. Depuración

Una habilidad que debes desarrollar cuando programas es siempre preguntarte a ti mismo, “¿Qué podría fallar aquí?” o alternativamente, “¿Qué cosa ilógica podría hacer un usuario para hacer fallar nuestro (aparentemente) perfecto programa?”

Por ejemplo, observa el programa que utilizamos para demostrar el bucle `while` en el apartado de iteraciones:

```
linea = input('> ')
while linea != 'fin':
    if linea[0] != '#' :
        print(linea)
    linea = input('> ')
print('¡Terminado!')

# Código: https://es.py4e.com/code3/copytildone2.py
```

Mira lo que pasa cuando el usuario introduce una línea vacía como entrada:

```
> hola a todos
hola a todos
> # no imprimas esto
> ¡imprime esto!
¡imprime esto!
>
Traceback (most recent call last):
  File "copytildone.py", line 3, in <module>
    if linea[0] != '#' :
IndexError: string index out of range
```

El código funciona bien hasta que se presenta una línea vacía. En ese momento no hay un carácter cero, por lo que obtenemos una traza de error (traceback). Existen dos soluciones a esto para convertir la línea tres en “segura”, incluso si la línea está vacía.

Una posibilidad es simplemente usar el método `startswith` que devuelve `False` si la cadena está vacía.

```
if linea.startswith('#'):
```

Otra forma segura es escribir una sentencia `if` utilizando el patrón *guardián* y asegurarse que la segunda expresión lógica es evaluada sólo cuando hay al menos un carácter en la cadena:

```
if len(linea) > 0 and linea[0] != '#':
```

# Fuente

- Página de Juan Jose Lozano Gomez sobre Python
- Python para todos
- Ejemplos con métodos de String
- Estructuras de datos
- Aprende con Alf

# * Práctica 3.0: Cadenas

**Ejercicios Nivel Básico y Medio**

# 3.1.-Est.Datos: Listas

## 3.1 Listas

Las listas son una de las estructuras de datos más versátiles en Python. A continuación, se describe a que nos referimos con estructuras de datos y se profundiza en las listas.

### 1. Estructuras de datos

Las **estructuras de datos en programación** son un modo de representar información en una computadora, aunque, además, cuentan con un comportamiento interno. ¿Qué significa? Que se rige por determinadas reglas/restricciones que han sido dadas por la forma en que está construida internamente.

¿Por qué es importante conocerlas? Cuando inicias en el mundo de la programación y te metes de lleno en el mundo de Python, las **estructuras de datos** son fundamentales. Conocer sobre listas, árboles y grafos te permitirá organizar mejor la información y crear código más eficiente. Además, es clave para mejorar tus habilidades técnicas y enfrentarte con éxito a cada reto en tus futuras entrevistas laborales.

En Python estas estructuras vienen definidas en la biblioteca estándar de Python

#### 1.1. ¿Para qué sirven las estructuras de datos?

En el ámbito de la informática, las **estructuras de datos** son aquellas que nos permiten, como desarrolladores, organizar la información de manera eficiente, y en definitiva diseñar la solución correcta para un determinado problema.

Ya sean las más utilizadas comúnmente - como las **variables**, **arrays**, **conjuntos ** o **clases** -** o las diseñadas para un propósito específico - **árboles**, **grafos**, **tablas**, etc., una **estructura de datos ** nos permite trabajar en un alto nivel de abstracción almacenando información para luego acceder a ella, modificarla y manipularla.

#### 1.2. ¿Cuáles son los tipos de estructuras de datos?

Primero, debemos diferenciar entre **estructura de dato estática** y **estructura de dato dinámica**

##### 1.2.1. Estructura de datos estáticas

Las **estructuras de datos estáticas** son aquellas en las que el tamaño ocupado en memoria se define antes de que el programa se ejecute y no puede modificarse dicho tamaño durante la ejecución del programa, mientras que una **estructura de datos dinámica** es aquella en la que el tamaño ocupado en memoria puede modificarse durante la ejecución del programa.

Cada tipo de estructura dependerá del tipo de aplicación que se requiera. Una **estructura de datos estáticas** típica son los **arrays**:

### ¿Qué es un array en programación?

Un **array** es un tipo de **dato estructurado** que permite almacenar un conjunto de datos homogéneo y ordenado, es decir, todos ellos del mismo tipo y relacionados. Su condición de *homogéneo*, indica que sus elementos están compuestos por el mismo tipo de dato, y su condición de *ordenado* hace que se pueda identificar la posición que ocupan desde el primer al último elemento que lo compone.

Los arrays de términos generales hacen referencia a menudo a los vectores y las matrices. Un vector es un array de 1 fila x n columnas (vector de fila) o 1 columna x n filas (vector de columna), mientras que una matriz es un array de m filas x n columnas.

### 1.2.2. Estructura de datos dinámicas

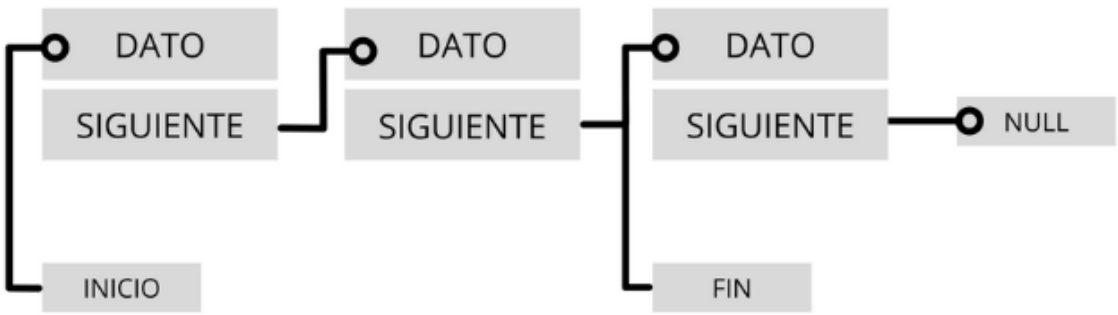
Por otro lado, en programación existen **estructuras de datos dinámicas**, es decir, una colección de elementos -nodos- que se utilizan para almacenar y organizar datos. A diferencia de un **array** que contiene espacio para almacenar un número fijo de elementos, una **estructura dinámica de datos** se amplía y contrae durante la ejecución del programa. Veamos algunos ejemplos:

### Estructura de datos lineales

Las **estructuras de datos lineales** son aquellas en las que los elementos ocupan lugares sucesivos en la estructura y cada uno de ellos tiene un único sucesor y un único predecesor, es decir, sus elementos están ubicados uno al lado del otro relacionados en forma lineal.

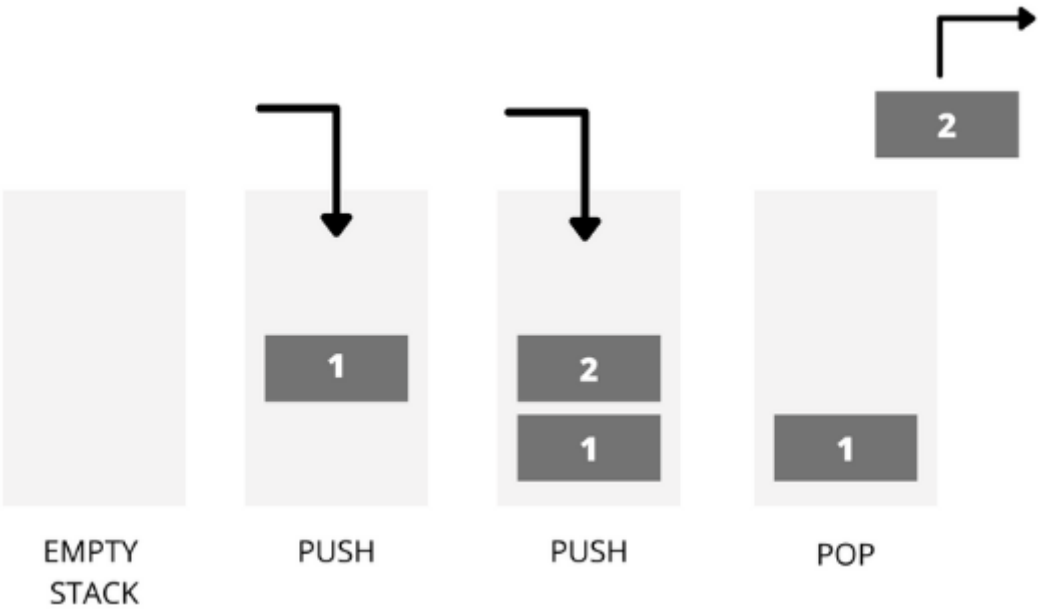
Hay tres tipos de **estructuras de datos lineales** listas enlazadas, pilas y colas.:

- **Listas enlazadas**: En las **estructuras de datos**, las listas enlazadas se construyen con elementos que están ubicados en una secuencia. Aquí, cada elemento se conecta con el siguiente a través de un enlace que contiene la posición del siguiente elemento. De este modo, teniendo la referencia del principio de la lista podemos acceder a todos los elementos de la misma. También existen las listas doblemente enlazadas, en las que cada nodo tiene dos enlaces, uno al nodo anterior y otro al siguiente.




Estructura de datos: Lista doblemente enlazada

- **Pila**: es un tipo especial de **lista lineal** dentro de las **estructuras de datos dinámicas** que permite almacenar y recuperar datos, siendo el modo de acceso a sus elementos de tipo LIFO (del inglés *Last In, First Out*, es decir, *último en entrar, primero en salir*). ¿Cómo funciona? A través de dos operaciones básicas: apilar (push), que coloca un objeto en la pila, y su operación inversa, desapilar (pop), que retira el último elemento apilado.



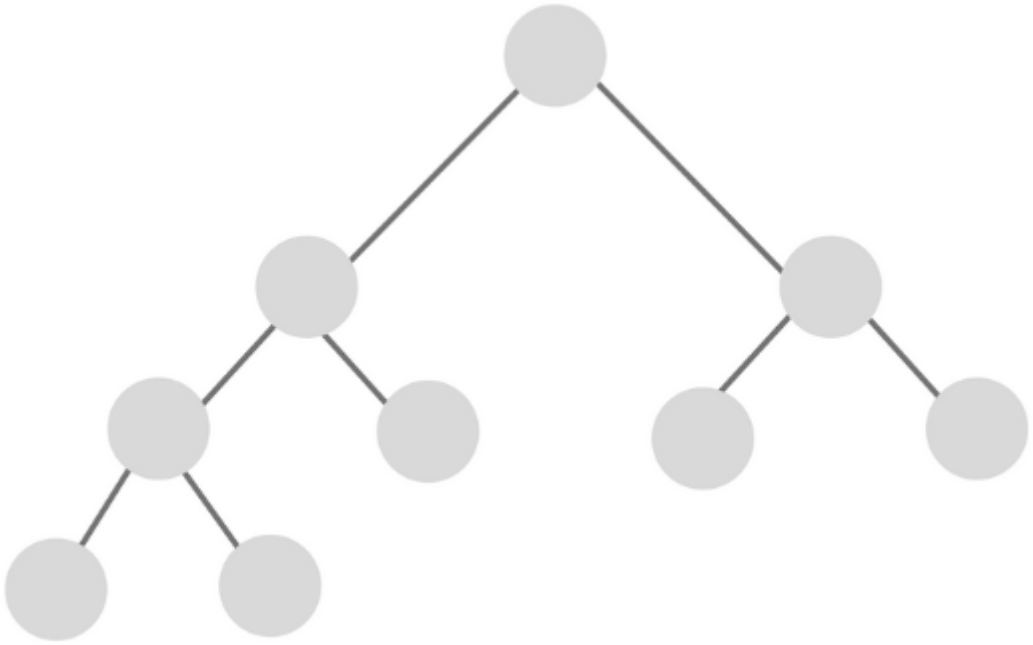

Estructura de datos: Pila

## Estructura de datos no lineales

Las **estructuras de datos no lineales**, también llamadas multienlazadas, son aquellas en las que cada elemento puede estar enlazado a cualquier otro componente. Es decir, cada elemento puede tener varios sucesores o varios predecesores.

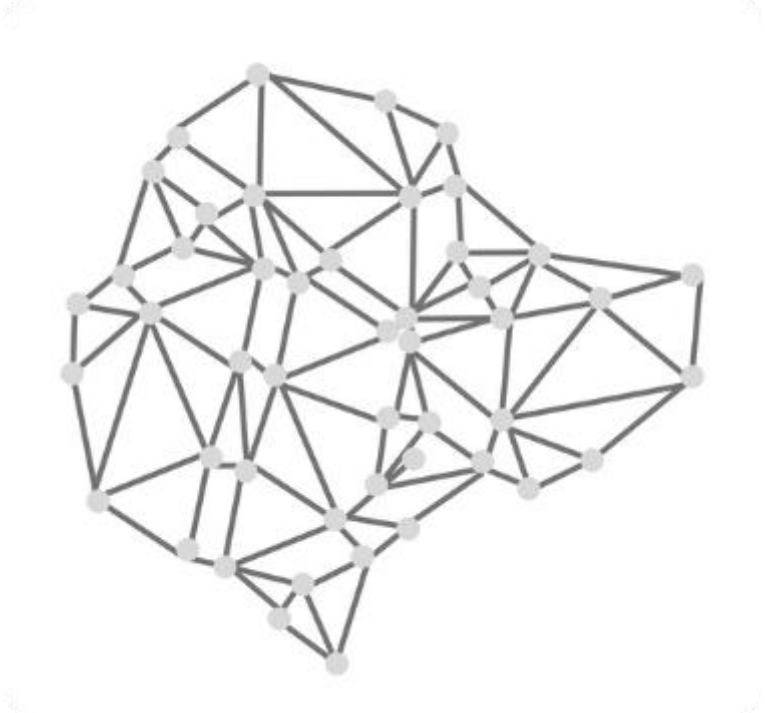
Entre ellos, destacamos dos tipos: Árboles, Grafos

- **Árboles**: En **estructura de datos**, los árboles consisten en una **estructura no lineal** que se utiliza para representar datos con una relación jerárquica en la que cada elemento tiene un único antecesor y puede tener varios sucesores.
  Los mismos se encuentran clasificados en:
  - **árbol general**, un árbol donde cada elemento puede tener un número ilimitado de sub árboles
  - **árboles binarios**, que son una estructura de datos homogénea, dinámica y no lineal en donde a cada elemento le pueden seguir como máximo dos nodos.



Estructura de datos: Árbol

- **Grafos**: Otro tipo **no lineal** de **estructura de datos en programación** son los **grafos**. Se trata de una estructura matemática formada por un conjunto de puntos — una estructura de datos — y un conjunto de líneas, cada una de las cuales une un punto a otro. Los puntos se llaman nodos o vértices del grafo y las líneas se llaman aristas o arcos.



Estructura de datos: Grafo

Las estructuras de datos son un aspecto clave a conocer en el mundo de la programación. Nos permiten mejorar nuestro código y habilidades técnicas, y en definitiva, resolver de manera eficiente problemas complejos.

## 2. Listas

Las listas en Python son un tipo **contenedor** compuesto, que **se usan para almacenar conjuntos de elementos relacionados** del mismo tipo o de tipos distintos.

Junto a las clases `tuple`, `range` y `str`, **son uno de los tipos de secuencia en Python**, con la particularidad de que son *mutables*. Esto último quiere decir que su contenido se puede modificar después de haber sido creada.

### 2.1. Crear una lista en Python

Para crear una lista en Python, simplemente hay que encerrar una secuencia de elementos separados por comas entre corchetes `[]`.

Por ejemplo, para crear una lista con los números del 1 al 10 se haría del siguiente modo:

```
>>> numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Como te decía, en python, las listas pueden almacenar elementos de distinto tipo. La siguiente lista también es válida:

```
>>> elementos = [3, 'a', 8, 7.2, 'hola']
```

Incluso pueden contener otros elementos compuestos, como objetos u otras listas:

```
>>> lista = [1, ['a', 'e', 'i', 'o', 'u'], 8.9, 'hola']
```

Las listas también se pueden crear usando el constructor de la clase, `list(iterable)`. En este caso, el constructor crea una lista cuyos elementos son los mismos y están en el mismo orden que los ítems del iterable. El objeto *iterable* puede ser o una secuencia, un contenedor que soporte la iteración o un objeto iterador.

Por ejemplo, el tipo *str* también es un tipo secuencia. Si pasamos un string al constructor `list()` creará una lista cuyos elementos son cada uno de los caracteres de la cadena:

```
>>> vocales = list('aeiou')
>>> vocales
>>> ['a', 'e', 'i', 'o', 'u']
```

Para terminar, a continuación, podemos ver dos alternativas para crear una lista vacía:

```
>>> lista_1 = []  # Opción 1
>>> lista_2 = list()  # Opción 2
```

## 2.2. Trabajar con los elementos de una lista

En Python, las listas son secuencias mutables, es decir, sus elementos pueden ser modificados (se pueden añadir nuevos ítems, actualizar o eliminar).

### 2.2.1 Cómo acceder a los elementos de una lista en Python

Para acceder a un elemento de una lista se utilizan los índices. **Un índice es un número entero que indica la posición de un elemento en una lista**. El primer elemento de una lista siempre comienza en el índice 0.

Por ejemplo, en una lista con 4 elementos, los índices de cada uno de los ítems serían 0, 1, 2 y 3.

```
>>> lista = ['a', 'b', 'd', 'i', 'j']
>>> lista[0]  # Primer elemento de la lista. Índice 0
>>> 'a'
>>> lista[3]  # Cuarto elemento de la lista. Índice 3
>>> 'i'
```

Si se intenta acceder a un índice que está fuera del rango de la lista, el intérprete lanzará la excepción `IndexError`. De igual modo, si se utiliza un índice que no es un número entero, se lanzará la excepción `TypeError`:

```
>>> lista = [1, 2, 3]  # Los índices válidos son 0, 1 y 2
>>> lista[8]
>>> Traceback (most recent call last):
>>> File "<input>", line 1, in <module>
>>> IndexError: list index out of range
>>> lista[1.0]
>>> Traceback (most recent call last):
>>> File "<input>", line 1, in <module>
>>> TypeError: list indices must be integers or slices, not float
```

Como hemos visto, las listas pueden contener otros elementos de tipo secuencia de forma anidada. Por ejemplo, una lista que uno de sus ítems es otra lista. Del mismo modo, se puede acceder a los elementos de estos tipos usando índices compuestos o anidados:

```
>>> lista = ['a', ['d', 'b'], 'z']
>>> lista[1][1]  # lista[1] hace referencia a la lista anidada
>>> 'b'
```

#### Acceso a los elementos usando un índice negativo

En Python está permitido usar índices negativos para acceder a los elementos de una secuencia. En este caso, el índice -1 hace referencia al último elemento de la secuencia, el -2 al penúltimo y así, sucesivamente:

```
>>> vocales = ['a', 'e', 'i', 'o', 'u']
>>> vocales[-1]
>>> 'u'
>>> vocales[-4]
>>> 'e'
```

### Acceso a un subconjunto de elementos

También es posible acceder a un subconjunto de elementos de una lista utilizando rangos en los índices. Esto es usando el operador `[:]`:

```
>>> vocales = ['a', 'e', 'i', 'o', 'u']
>>> vocales[2:3]  # Elementos desde el índice 2 hasta el índice 3-1
>>> ['i']
>>> vocales[2:4]  # Elementos desde el 2 hasta el índice 4-1
>>> ['i', 'o']
>>> vocales[:]  # Todos los elementos
>>> ['a', 'e', 'i', 'o', 'u']
>>> vocales[1:]  # Elementos desde el índice 1
>>> ['e', 'i', 'o', 'u']
>>> vocales[:3]  # Elementos hasta el índice 3-1
>>> ['a', 'e', 'i']
```

También es posible acceder a los elementos de una lista indicando un paso con el operador `[::]`:

```
>>> letras = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k']
>>> letras[::2]  # Acceso a los elementos de 2 en 2
>>> ['a', 'c', 'e', 'g', 'i', 'k']
>>> letras[1:5:2]  # Elementos del índice 1 al 4 de 2 en 2
>>> ['b', 'd']
>>> letras[1:6:3]  # Elementos del índice 1 al 5 de 3 en 3
>>> ['b', 'e']
```

## 2.2.2. Añadir elementos a una lista en Python

Tal y como te he adelantado, las listas son secuencias mutables, es decir, sus elementos pueden ser modificados (se pueden añadir nuevos ítems, actualizar o eliminar).

Para añadir un **nuevo elemento** a una lista se utiliza el método `append()` y para **añadir varios elementos**, el método `extend()`:

```
>>> vocales = ['a']
>>> vocales.append('e')  # Añade un elemento
>>> vocales
>>> ['a', 'e']
>>> vocales.extend(['i', 'o', 'u'])  # Añade un grupo de elementos
>>> vocales
>>> ['a', 'e', 'i', 'o', 'u']
```

También es posible utilizar el operador de **concatenación** `+` para unir dos listas en una sola. El resultado es una nueva lista con los elementos de ambas:

```
>>> lista_1 = [1, 2, 3]
>>> lista_2 = [4, 5, 6]
>>> nueva_lista = lista_1 + lista_2
>>> nueva_lista
>>> [1, 2, 3, 4, 5, 6]
```

Por otro lado, el operador `*` repite el contenido de una lista `n` veces:

```
>>> numeros = [1, 2, 3]
```

```
>>> numeros *= 3
>>> numeros
>>> [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Y para terminar esta sección, indicarte que también es posible añadir un elemento en una posición concreta de una lista con el método `insert(índice, elemento)`. Los elementos cuyo índice sea mayor a `índice` se desplazan una posición a la derecha:

```
>>> vocales = ['a', 'e', 'u']
>>> vocales.insert(2, 'i')
>>> vocales
>>> ['a', 'e', 'i', 'u']
```

### 2.2.3. Modificar elementos de una lista

Es posible **modificar** un elemento de una lista en Python con el operador de asignación =. Para ello, lo único que necesitas conocer es el índice del elemento que quieres modificar o el rango de índices:

```
>>> vocales = ['o', 'o', 'o', 'o', 'u']

# Actualiza el elemento del índice 0
>>> vocales[0] = 'a'
>>> vocales
>>> ['a', 'o', 'o', 'o', 'u']

# Actualiza los elementos entre las posiciones 1 y 2
>>> vocales[1:3] = ['e', 'i']
>>> vocales
>>> ['a', 'e', 'i', 'o', 'u']
```

### 2.2.4. Eliminar un elemento de una lista en Python

En Python se puede **eliminar** un elemento de una lista de varias formas.

Con la sentencia `del` se puede eliminar un elemento a partir de su índice:

```
# Elimina el elemento del índice 1
>>> vocales = ['a', 'e', 'i', 'o', 'u']
>>> del vocales[1]
>>> vocales
>>> ['a', 'i', 'o', 'u']

# Elimina los elementos con índices 2 y 3
>>> vocales = ['a', 'e', 'i', 'o', 'u']
>>> del vocales[2:4]
>>> vocales
>>> ['a', 'e', 'u']

##### Elimina todos los elementos
>>> del vocales[:]
>>> vocales
>>> []
```

Además de la sentencia `del`, podemos usar los métodos `remove()` y `pop([i])`. `remove()` elimina la primera ocurrencia que se encuentre del elemento en una lista. Por su parte, `pop([i])` obtiene el elemento cuyo índice sea igual a `i` y lo elimina de la lista. Si no se especifica ningún índice, recupera y elimina el último elemento.

```
>>> letras = ['a', 'b', 'k', 'a', 'v']

# Elimina la primera ocurrencia del carácter a
>>> letras.remove('a')
>>> letras
>>> ['b', 'k', 'a', 'v']

# Obtiene y elimina el último elemento
>>> letras.pop()
>>> 'v'
>>> letras
>>> ['b', 'k', 'a']
```

Finalmente, es posible eliminar todos los elementos de una lista a través del método `clear()`:

```
>>> letras = ['a', 'b', 'c']
>>> letras.clear()
>>> letras
>>> []
```

El código anterior sería equivalente a `del letras[:]`.

### 2.3. Patrones de uso

Algunos patrones de uso de las listas son los siguientes:

#### 2.3.1. Longitud (len) de una lista en Python

Como cualquier tipo secuencia, para conocer la longitud de una lista en Python se hace uso de la función `len()`. Esta función devuelve el número de elementos de una lista:

```
>>> vocales = ['a', 'e', 'i', 'o', 'u']
>>>
>>> len(vocales)
>>> 5
```

#### 2.3.2. Recorrer una lista - `for` list Python

Como vimos en las unidades anteriores, se puede usar el bucle for en Python para recorrer los elementos de una secuencia `String`. En nuestro caso, para recorrer una lista en Python utilizaríamos la siguiente estructura, similar a la que ya hemos visto:

```
>>> colores = ['azul', 'blanco', 'negro']
>>> for color in colores:
>>>     print(color)
>>> azul
>>> blanco
>>> negro
```

#### 2.3.3. Cómo saber si un elemento está en una lista en Python

Como vimos anteriormente con las cadenas, para saber si un elemento carácter está contenido en una lista de caracteres, se utiliza el operador de pertenencia `in`. En las listas funciona exactamente igual:

```
>>> vocales = ['a', 'e', 'i', 'o', 'u']
>>> if 'a' in vocales:
>>> ...     print('Sí')
>>> ...
>>> Sí
>>> if 'b' not in vocales:
>>> ...     print('No')
>>> ...
>>> No
```

### 2.3.4. Ordenar una lista en Python - `sort` list Python

Las listas son secuencias no ordenadas. Esto quiere decir que sus elementos siempre se devuelven en el mismo orden en que fueron añadidos.

No obstante, es posible ordenar los elementos de una lista con el método `sort()`. El método `sort()` ordena los elementos de la lista utilizando únicamente el operador < y modifica la lista actual (no se obtiene una nueva lista):

```
# Lista desordenada de números enteros
>>> numeros = [3, 2, 6, 1, 7, 4]


# Identidad del objeto numeros
>>> id(numeros)
>>> 4475439216

# Se llama al método sort() para ordenar los elementos de la lista
>>> numeros.sort()
>>> numeros
>>> [1, 2, 3, 4, 6, 7]

# Se comprueba que la identidad del objeto numeros es la misma
>>> id(numeros)
>>> 4475439216
```

### 2.3.5. Copia de listas

Existen **dos formas de copiar listas**:

- **Copia por referencia** `l1 = l2`: Asocia a la variable `l1` la misma lista que tiene asociada la variable `l2`, es decir, ambas variables apuntan a la misma dirección de memoria. Cualquier cambio que hagamos a través de `l1` o `l2` afectará a la misma lista.
- **Copia por valor** `l1 = list(l2)`: Crea una copia de la lista asociada a `l2` en una dirección de memoria diferente y se la asocia a `l1`. Las variables apuntan a direcciones de memoria diferentes que contienen los mismos datos. Cualquier cambio que hagamos a través de `l1` no afectará a la lista de `l2` y viceversa.

```
>>> a = [1, 2, 3]
>>> # copia por referencia
>>> b = a
>>> b
[1, 2, 3]
>>> b.remove(2)
>>> b
[1, 3]
```

```
>>> a
[1, 3]
>>> a = [1, 2, 3]
>>> # copia por referencia
>>> b = list(a)
>>> b
[1, 2, 3]
>>> b.remove(2)
>>> b
[1, 3]
>>> a
[1, 2, 3]
```

### 2.4. Listado de métodos de la clase list

Finalizamos mostrando la lista completa de métodos de la clase `list`. Algunos de ellos ya se han mencionado en las secciones anteriores.

| Método | Descripción |
|---|---|
| `append()` | Añade un nuevo elemento al final de la lista. |
| `extend()` | Añade un grupo de elementos (iterables) al final de la lista. |
| `insert(indice, elemento)` | Inserta un elemento en una posición concreta de la lista. |
| `remove(elemento)` | Elimina la primera ocurrencia del elemento en la lista. |
| `pop([i])` | Obtiene y elimina el elemento de la lista en la posición i. Si no se especifica, obtiene y elimina el último elemento. |
| `clear()` | Borra todos los elementos de la lista. |
| `index(elemento)` | Obtiene el índice de la primera ocurrencia del elemento en la lista. Si el elemento no se encuentra, se lanza la excepción`ValueError`. |
| `count(elemento)` | Devuelve el número de ocurrencias del elemento en la lista. |
| `sort()` | Ordena los elementos de la lista utilizando el operador`<`. |
| `reverse()` | Obtiene los elementos de la lista en orden inverso. |
| `copy()` | Devuelve una copia poco profunda de la lista. |

## 3. Depuración

El uso descuidado de listas (y otros objetos mutables) puede llevar a largas horas de depuración. Aquí están algunos de los errores más comunes y las formas de evitarlos:

### 3.1. Revisa los métodos

No olvides que la mayoría de métodos de listas modifican el argumento y regresan `None`. ***Esto es lo opuesto a los métodos de cadenas***, que regresan una nueva cadena y dejan la original sin modificar.

Si estás acostumbrado a escribir código de cadenas como este:

```
palabra = palabra.strip()
```

Estás propenso a escribir código de listas como este:

```
t = t.sort()            # ¡EQUIVOCADO!
```

Debido a que `sort` regresa `None`, la siguiente operación que hagas con `t` es probable que falle.

Antes de usar métodos y operadores de listas, deberías leer la documentación cuidadosamente y después probarlos en modo interactivo. Los métodos y operadores que las listas comparten con otras secuencias (como cadenas) están documentados en:

docs.python.org/library/stdtypes.html#common-sequence-operations

Los métodos y operadores que solamente aplican a secuencias mutables están documentados en:

docs.python.org/library/stdtypes.html#mutable-sequence-types

### 3.2. Elige un estilo y apégate a él.

Parte del problema con listas es que hay demasiadas formas de hacer las cosas. Por ejemplo, para remover un elemento de una lista, puedes utilizar `pop`, `remove`, `del`, o incluso una asignación por rebanado. Para agregar un elemento, puedes utilizar el método `append` o el operador `+`. Pero no olvides que esos también son correctos:

```
t.append(x)
t = t + [x]
```

Y esos son incorrectos:

```
t.append([x])           # ¡EQUIVOCADO!
t = t.append(x)         # ¡EQUIVOCADO!
t + [x]                 # ¡EQUIVOCADO!
t = t + x               # ¡EQUIVOCADO!
```

***Prueba cada uno de esos ejemplos en modo interactivo para asegurarte que entiendes lo que hacen. Nota que solamente la última provoca un error en tiempo de ejecución (runtime error); los otros tres son válidos, pero hacen la función equivocada.***

### 3.3. Hacer copias para evitar alias.

Si quieres utilizar un método como `sort` que modifica el argumento, pero necesitas mantener la lista original también, puedes hacer una copia.

```
orig = t[:]
t.sort()
```

En este ejemplo podrías también usar la función interna `sorted`, la cual regresa una lista nueva y ordenada, y deja la original sin modificar. ¡*Pero en ese caso deberías evitar usar* `sorted` *como un nombre de variable!*

### 3.4. Listas, `split`, y ARCHIVOS

Cuando leemos y analizamos archivos, hay muchas oportunidades de encontrar entradas que pueden hacer fallar a nuestro programa, así que es una buena idea revisar el patrón *guardián* cuando escribimos programas que leen a través de un archivo y buscan una "aguja en un pajar". Vamos a revisar nuestro programa que busca por el día de la semana en las líneas que contienen "from" en el archivo":

```
From stephen.marquard@uct.ac.za Sat Jan  5 09:14:16 2008
```

Puesto que estamos dividiendo esta línea en palabras, podríamos apañarnos con el uso de `startswith` y simplemente buscar la primera palabra de la línea para determinar si estamos interesados en esa línea o no. Podemos saltarnos las líneas que no tienen "From" como primer palabra, tal como sigue:

```
manejador = open('mbox-short.txt')
for linea in manejador:
    palabras = linea.split()
    if palabras[0] != 'From' : continue # Hay otra forma, evita
continue
    print(palabras[2])
```

Esto se ve mucho más simple y ni siquiera necesitamos hacer `rstrip` para borrar el salto de línea al final del archivo. Pero, ¿es mejor?

```
python search8.py
Sat
Traceback (most recent call last):
  File "search8.py", line 5, in <module>
    if palabras[0] != 'From' : continue
IndexError: list index out of range
```

De alguna manera funciona y vemos el día de la primera línea (Sat), pero luego el programa falla con un error. ¿Qué fue lo que falló? ¿Qué datos estropearon e hicieron fallar a nuestro elegante, inteligente, y muy Pythónico programa?

Puedes mirar el código por un largo tiempo y tratar de resolverlo o preguntar a alguien más, pero el método más rápido e inteligente es agregar una sentencia `print`. El mejor lugar para agregar la sentencia `print` es justo antes de la línea donde el programa falló, e imprimir los datos que parece que causan la falla.

Ahora bien, este método podría generar muchas líneas de salida, pero al menos tendrás inmediatamente alguna pista de cuál es el problema. Así que agregamos un print a la variable `palabras` justo antes de la línea cinco. Incluso podemos agregar un prefijo "Depuración:" a la línea de modo que mantenemos nuestra salida regular separada de la salida de mensajes de depuración.

```
for linea in manejador:
    palabras = line.split()
    print('Depuración:', palabras)
    if palabras[0] != 'From' : continue
    print(palabras[2])
```

Cuando ejecutamos el programa, se generan muchos mensajes de salida en la pantalla, pero al final, vemos nuestra salida de depuración y el mensaje de error, de modo que sabemos qué sucedió justo antes del error.

```
Depuración: ['X-DSPAM-Confidence:', '0.8475']
Depuración: ['X-DSPAM-Probability:', '0.0000']
Depuración: []
Traceback (most recent call last):
  File "search9.py", line 6, in <module>
    if palabras[0] != 'From' : continue
IndexError: list index out of range
```

Cada línea de depuración imprime la lista de palabras que obtuvimos cuando la función `split` dividió la línea en palabras. Cuando el programa falla, la lista de palabras está vacía `[]`. Si abrimos el archivo en un editor de texto y miramos el archivo, en ese punto se ve lo siguiente:

```
X-DSPAM-Result: Innocent
X-DSPAM-Processed: Sat Jan  5 09:14:16 2008
X-DSPAM-Confidence: 0.8475
X-DSPAM-Probability: 0.0000

Details: http://source.sakaiproject.org/viewsvn/?view=rev&rev=39772
```

¡El error ocurre cuando nuestro programa encuentra una línea vacía! Por supuesto, hay "cero palabras" en una lista vacía. ¿Por qué no pensamos en eso cuando estábamos escribiendo el código? Cuando el código busca la primera palabra (`palabras[0]`) para revisar si coincide con "From", obtenemos un error "index out of range" (índice fuera de rango).

Este es, por supuesto, el lugar perfecto para agregar algo de *código guardián* para evitar revisar si la primera palabra no existe. Hay muchas maneras de proteger este código; vamos a optar por revisar el número de palabras que tenemos antes de mirar la primera palabra:

```
manejador = open('mbox-short.txt')
contador = 0
```

```
for linea in manejador:
    palabras = linea.split()
    # print 'Depuración:', palabras
    if len(palabras) == 0 : continue
    if palabras[0] != 'From' : continue
    print(palabras[2])
```

Posteriormente comentaremos la sentencia de depuración en vez de borrarla, en caso de que nuestra modificación falle y tengamos que depurar de nuevo. Luego, agregamos una **sentencia guardián** que revisa si tenemos cero palabras, y si así fuera, saltaremos a la siguiente línea en el archivo.

Podemos pensar en las dos sentencias `continue` (seguro que encuetnras otra forma que no sea con `continue`) que se usan para solo procesar las líneas que son "interesantes" en nuestro proceso. Una línea que no tenga palabras "no es interesante" para nosotros así que saltamos a la siguiente línea. Una línea que no tenga "From" como su primera palabra tampoco nos interesa así que la saltamos.

El programa modificado se ejecuta con éxito, así que quizás es correcto. Nuestra sentencia guardián se asegura de que `palabras[0]` nunca falle, pero quizá no sea suficiente. Cuando estamos programando, siempre debemos pensar, "¿qué podría salir mal?"

# Fuente

- Pagina de Juan Jose Lozano Gomez sobre Python
- Estructuras de datos
- Python para todos
- Aprende con Alf

# 3.2.-Est.Datos: Tuplas

## 3.2. Tuplas

Otra estructura de datos que nos ofrece Python son las tuplas. Vamos a ver qué son y cómo trabajar con ellas.

### 1. Que es una tupla

La clase `tuple` en Python es un tipo **contenedor** compuesto, que **en un principio se pensó para almacenar grupos de elementos heterogéneos**, aunque también puede contener elementos homogéneos. Se pueden considerar como **listas inmutables**. Se definen entre paréntesis y los elementos se separan por comas.

Junto a las clases *list* y *range*, **es uno de los tipos de secuencia en Python**, con la particularidad de que son *inmutables*. Esto último quiere decir que su contenido **NO** se puede modificar después de haber sido creada.

*En general, para crear una tupla en Python simplemente hay que definir una secuencia de elementos separados por comas. No es obligarorio encerrar los elementos entre paréntesis, aunque es una buena práctica hacerlo.*

Por ejemplo, para crear una tupla con los números del 1 al 5 se haría del siguiente modo:

```
>>> numeros = 1, 2, 3, 4, 5
```

La clase `tuple` también puede almacenar elementos de distinto tipo:

```
>>> elementos = 3, 'a', 8, 7.2, 'hola'
```

Incluso pueden contener otros elementos compuestos y objetos, como listas, otras tuplas, etc.:

```
>>> tup = 1, ['a', 'e', 'i', 'o', 'u'], 8.9, 'hola'
```

A continuación, se ven las diferentes formas que existen de crear una tupla en Python:

- Para crear una tupla vacía, usa paréntesis `()` o el constructor de la clase `tuple()` sin parámetros.
- Para crear una tupla con un único elemento: `elem,` o `(elem, )`. Observa que siempre se añade una coma.
- Para crear una tupla de varios elementos, sepáralos con comas: `a, b, c` o `(a, b, c)`.
- Las tuplas también se pueden crear usando el constructor de la clase, `tuple(iterable)`. En este caso, el constructor crea una tupla cuyos elementos son los mismos y están en el mismo orden que los ítems del iterable. El objeto *iterable* puede ser una secuencia, un contenedor que soporte la iteración o un objeto iterador.

**IMPORTANTE:** El hecho que determina que una secuencia de elementos sea una tupla es la coma, no los paréntesis. Los paréntesis son opcionales y solo se necesitan para crear una tupla vacía o para evitar ambigüedades.

```
# Aquí, a, b y c no son una tupla, sino tres argumentos con
# los que se llama a la función "una_funcion"
>>> una_funcion(a, b, c)
# Aquí, a, b y c son tres elementos de una tupla. Esta tupla,
# es el único argumento con el que se invoca a la
# función "una_funcion"
>>> una_funcion((a, b, c))
```

La forma de crear una tupla sin paréntesis es conocida como *tuple packing* (algo así como empaquetado de tuplas).

## 2. Diferencia entre tuplas y listas.

Una lista no es lo mismo que una tupla. Ambas son un conjunto ordenado de valores, en donde este último puede ser cualquier objeto: un número, una cadena, una función, una clase, una instancia, etc. La diferencia es que las listas presentan una serie de funciones adicionales que permiten un amplio manejo de los valores que contienen. Basándonos en esta definición, puede decirse que las listas son dinámicas, mientras que las tuplas son estáticas.

La principal diferencia entre las listas y las tuplas de Python, y el motivo por el que muchos usuarios solamente utilizar listas, es que **las listas son mutables** mientras que **las tuplas son inmutables**. ¿Pero qué significa ser mutable o no? Básicamente un objeto mutable se puede modificar una vez creado mientras que uno que no lo es no. Así el contenido de las listas se puede modificar durante la ejecución del programa mientras para las tuplas no es posible alterar su contenido. Las tuplas se podrán usar como las listas teniendo en cuenta su inmutabilidad.

El hecho de ser mutable tiene además otras consecuencias. Para ser mutables las listas se almacena en dos bloques de memoria, mientras que las tuplas solo necesitan uno. Lo que provoca que las tuplas ocupen menos memoria que las listas. Además, por el hecho de no ser mutables, es más rápido manejar tuplas que listas. Debido a esto, hay que tener en cuenta lo anterior para elegir en nuestros algoritmos el tipo que mejor se adapte. En el caso de que no sea necesario modificar el contenido de los datos la mejor opción es la tupla, ya que es ocupa menos memoria y es más rápida. En el resto de los casos la mejor opción será utilizar listas.

## 3. Trabajar con tuplas en Python

Las tuplas en Python son objetos inmutables, lo que significa que una vez creadas no se pueden modificar. A continuación, se muestran algunas operaciones básicas que se pueden realizar con las tuplas en Python.

### 3.1. Cómo acceder a los elementos de una tupla en Python

Para acceder a un elemento de una tupla se utilizan los índices. **Un índice es un número entero que indica la posición de un elemento en una tupla**. El primer elemento de una tupla siempre comienza en el índice 0.

Por ejemplo, en una tupla con 3 elementos, los índices de cada uno de los ítems serían 0, 1 y 2.

```
>>> tupla = ('a', 'b', 'd')
>>> tupla[0]  # Primer elemento de la tupla. Índice 0
'a'
>>> tupla[1]  # Segundo elemento de la tupla. Índice 1
'b'
```

Si se intenta acceder a un índice que está fuera del rango de la tupla, el intérprete lanzará la excepción `IndexError`. De igual modo, si se utiliza un índice que no es un número entero, se lanzará la excepción `TypeError`:

```
>>> tupla = 1, 2, 3  # Los índices válidos son 0, 1 y 2
>>> tupla[8]
Traceback (most recent call last):
File "<input>", line 1, in <module>
IndexError: tuple index out of range
>>> tupla[1.0]
Traceback (most recent call last):
File "<input>", line 1, in <module>
TypeError: tuple indices must be integers or slices, not float
>
```

### 3.1.1. Acceso a los elementos usando un índice negativo

Al igual que ocurre con las listas (y todos los tipos secuenciales), está permitido usar índices negativos para acceder a los elementos de una tupla. En este caso, el índice -1 hace referencia al último elemento de la secuencia, el -2 al penúltimo y así, sucesivamente:

```
>>> bebidas = ('agua', 'café', 'batido', 'sorbete')
>>> bebidas[-1]
'sorbete'
>>> bebidas[-3]
'café'
```

### 3.1.2. Acceso a un subconjunto de elementos

También es posible acceder a un subconjunto de elementos de una tupla utilizando el operador `[:]`:

```
>>> vocales = 'a', 'e', 'i', 'o', 'u'
>>> vocales[2:3]  # Elementos desde el índice 2 hasta el índice 3-1
('i',)
>>> vocales[2:4]  # Elementos desde el 2 hasta el índice 4-1
('i', 'o')
>>> vocales[:]  # Todos los elementos
('a', 'e', 'i', 'o', 'u')
>>> vocales[1:]  # Elementos desde el índice 1
('e', 'i', 'o', 'u')
>>> vocales[:3]  # Elementos hasta el índice 3-1
('a', 'e', 'i')
```

O indicando un salto entre los elementos con el operador `[::]`:

```
>>> pares = 2, 4, 6, 8, 10, 12, 14
>>> pares[::2]  # Acceso a los elementos de 2 en 2
(2, 6, 10, 14)
>>> pares[1:5:2]  # Elementos del índice 1 al 4 de 2 en 2
(4, 8)
>>> pares[1:6:3]  # Elementos del índice 1 al 5 de 3 en 3
(4, 10)
```

### 3.2. Modificar una tupla en Python

Como hemos dicho ya, las tuplas son objetos inmutables. No obstante, las tuplas pueden contener objetos u otros elementos de tipo secuencia, por ejemplo, una lista. Estos objetos, si son mutables, sí se pueden modificar:

```
>>> tupla = (1, ['a', 'b'], 'hola', 8.2)
>>> tupla[1].append('c')  # tupla[1] hace referencia a la lista
>>> tupla
(1, ['a', 'b', 'c'], 'hola', 8.2)
```

## 4. Patrones de uso

### 4.1. Longitud (len) de una tupla en Python

Como cualquier tipo secuencia, para conocer la longitud de una tupla en Python se hace uso de la función `len()`. Esta función devuelve el número de elementos de una tupla:

```
>>> vocales = ('a', 'e', 'i', 'o', 'u')
>>> len(vocales)
5
```

### 4.2. Recorrer una tupla - `for` tuple Python

El bucle `for` en Python es una de las estructuras ideales para iterar sobre los elementos de una secuencia. Para recorrer una tupla en Python utiliza la siguiente estructura:

```
>>> colores = 'azul', 'blanco', 'negro'
>>> for color in colores:
...   print(color)
azul
blanco
negro
```

### 4.3. Cómo saber si un elemento está en una tupla en Python

Como hemos visto en otras unidades, para saber si un elemento está contenido en una tupla, se utiliza el operador de pertenencia `in`:

```
>>> colores = 'azul', 'blanco', 'negro'
>>> if 'azul' in colores:
...     print('Sí')
...
Sí
```

```
>>> if 'verde' not in colores:
...     print('No')
...
No
```

### 4.4. tuple unpacking

El concepto conocido como *tuple unpacking* (desempaquetado de una tupla) se puede aplicar sobre cualquier objeto de tipo secuencia, aunque se usa mayoritariamente con las tuplas, y consiste en lo siguiente:

```
>>> bebidas = 'agua', 'café', 'batido'
>>> a, b, c = bebidas
>>> a
'agua'
>>> b
'café'
>>> c
'batido'
```

Como puedes apreciar, es un tipo de asignación múltiple. Requiere que haya tantas variables a la izquierda del operador de asignación `=` como elementos haya en la secuencia.

## 5. Listado de métodos de la clase tuple en Python

Para terminar, se muestran los métodos de la clase `tuple` en Python, que son los métodos definidos para cualquier tipo secuencial:

| Método | Descripción |
|---|---|
| `index(elemento)` | Obtiene el índice de la primera ocurrencia del elemento en la tupla. Si el elemento no se encuentra, se lanza la excepción `ValueError`. |
| `count(elemento)` | Devuelve el número de ocurrencias del elemento en la tupla. |

# Fuente

- Pagina de Juan Jose Lozano Gomez sobre Python
- Estructuras de datos
- Python para todos
- Aprende con Alf

# * Práctica 3.1: Listas y Tuplas

# 3.3.-Est.Datos: Diccionarios

## 3.3. Diccionarios

Los diccionarios son una estructura de datos que permite almacenar un conjunto de pares clave-valor. Cada clave es única y está asociada a un valor. Los diccionarios son una estructura de datos muy utilizada en Python y en otros lenguajes de programación. En este apartado vamos a ver cómo se crean y se utilizan los diccionarios en Python.

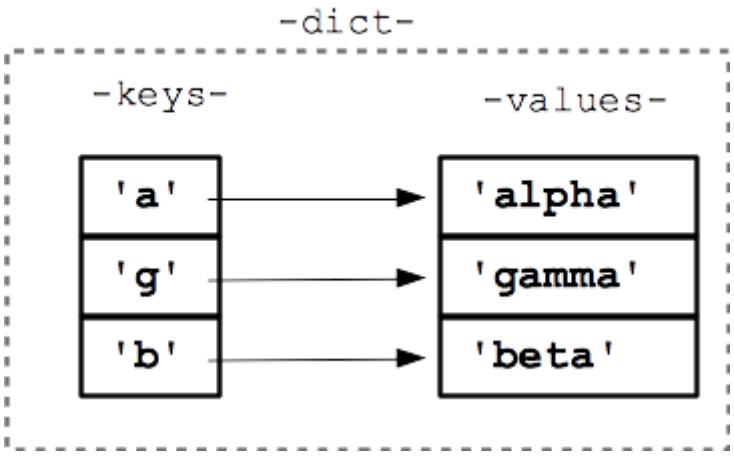
### 1. Introducción

La clase `dict` de Python es un tipo mapa que asocia claves a valores. A diferencia de los tipos secuenciales `list`, `tuple`, `range` o `str`, que son indexados por un índice numérico, los diccionarios son indexados por claves. Estas claves siempre deben ser de un tipo inmutable, concretamente un tipo *hashable*.

**NOTA:** Un objeto es *hashable* si tiene un valor de *hash* que no cambia durante todo su ciclo de vida. En principio, los objetos que son instancias de clases definidas por el usuario son *hashables*. También lo son la mayoría de tipos inmutables definidos por Python (`int`,`float` o `str`).

Piensa siempre en un diccionario como un contenedor de pares *clave*: *valor*, en el que la clave puede ser de cualquier tipo hashable y es única en el diccionario que la contiene. Generalmente, se suelen usar como claves los tipos `int` y `str` aunque, como se ha comentado, cualquier tipo *hashable* puede ser una clave.

**Ten cuenta que no todos los objetos pueden ser clave, solo los hashables**




Estructura de datos: Diccionario

**Las principales operaciones** que se suelen realizar con diccionarios **son almacenar un valor asociado a una clave y recuperar un valor a partir de una clave**. Esta es la esencia de los diccionarios y es aquí donde son realmente importantes. **En un diccionario, el acceso a un elemento a partir de una clave es una operación**

**realmente rápida, eficaz y que consume pocos recursos** si lo comparamos con cómo lo haríamos con otros tipos de datos.

Otras características a resaltar de los diccionarios:

- **Es un tipo mutable**: su contenido se puede modificar después de haber sido creado.
- **Es un tipo ordenado**: en el sentido de que preserva el orden en que se insertan los pares *clave: valor*.
- **Es un tipo dinámico**: no es necesario declarar el tamaño del diccionario antes de usarlo.
- **Es un tipo heterogéneo**: las claves y los valores pueden ser de cualquier tipo y mezclarse en un mismo diccionario.

## 2. Trabajar con diccionarios en Python

Los diccionarios en Python son un tipo de datos muy versátil y potente. Se pueden crear, modificar y eliminar elementos de un diccionario, así como acceder a sus elementos de distintas formas. A continuación, se describen las operaciones más comunes que se pueden realizar con diccionarios en Python.

### 2.1. Cómo crear un diccionario

En Python hay varias formas de crear un diccionario. Las veremos todas a continuación.

La más simple es encerrar una secuencia de pares *clave: valor* separados por comas entre llaves `{}`

```
>>> d = {1: 'hola', 89: 'Pythonista', 'a': 'b', 'c': 27}
```

En el diccionario anterior, los enteros `1` y `89` y las cadenas `'a'` y `'c'` son las claves. Como ves, se pueden mezclar claves y valores de distinto tipo sin problema.

Para crear un **diccionario vacío**:

- Asigna a una variable el valor `{}`.
- Usar el constructor de la clase `dict()`, aunque este constructor se puede de varias maneras:
    - **Sin parámetros** . Esto creará un diccionario vacío.
    - Con pares *clave: valor* encerrados entre llaves.
    - **Con argumentos con nombre**. El nombre del argumento será la clave en el diccionario. En este caso, las claves solo pueden ser identificadores válidos y mantienen el orden en el que se indican. No se podría, por ejemplo, tener números enteros como claves. Para utilizar los números enteros como claves, se debe usar otra forma de crear los diccionarios.
    - **Pasando un iterable**. En este caso, cada elemento del iterable debe ser también un iterable con solo dos elementos. El primero se toma como clave del diccionario y el segundo como valor. Si la clave aparece varias veces, el valor que prevalece es el último.

Veamos un ejemplo con todo lo anterior. Vamos a crear el mismo diccionario de todos los modos que te he explicado:

```
# 1. Pares clave: valor encerrados entre llaves
>>> d = {'uno': 1, 'dos': 2, 'tres': 3}
>>> d
{'uno': 1, 'dos': 2, 'tres': 3}

# 2. Argumentos con nombre
>>> d2 = dict(uno=1, dos=2, tres=3)
>>> d2
{'uno': 1, 'dos': 2, 'tres': 3}

# 3. Pares clave: valor encerrados entre llaves
>>> d3 = dict({'uno': 1, 'dos': 2, 'tres': 3})
>>> d3
{'uno': 1, 'dos': 2, 'tres': 3}

# 4. Iterable que contiene iterables con dos elementos
>>> d4 = dict([('uno', 1), ('dos', 2), ('tres', 3)])
>>> d4
{'uno': 1, 'dos': 2, 'tres': 3}

# 5. Diccionario vacío
>>> d5 = {}
>>> d5
{}

# 6. Diccionario vacío usando el constructor
>>> d6 = dict()
>>> d6
{}
```

### 2.2. Cómo acceder a los elementos de un diccionario en Python

Acceder a un elemento de un diccionario es una de las principales operaciones por las que existe este tipo de dato. El acceso a un valor se realiza mediante indexación de la clave. Para ello, simplemente encierra entre corchetes la clave del elemento `d[clave]`. En caso de que la clave no exista, se lanzará la excepción `KeyError`.

```
>>> d = {'uno': 1, 'dos': 2, 'tres': 3}
>>> d['dos']
2
>>> d[4]
Traceback (most recent call last):
File "<input>", line 1, in <module>
KeyError: 4
>
```

La clase `dict` también ofrece el método `get(clave [, valor por defecto])`. Este método devuelve el valor correspondiente a la clave `clave`. En caso de que la clave no exista **no lanza ningún error**, sino que devuelve el segundo argumento `valor por defecto`. Si no se proporciona este argumento, se devuelve el valor `None`.

```
>>> d = {'uno': 1, 'dos': 2, 'tres': 3}
>>> d.get('uno')
1
```

```
# Devuelve 4 como valor por defecto si no encuentra la clave
>>> d.get('cuatro', 4)
4

# Devuelve None como valor por defecto si no encuentra la clave
>>> a = d.get('cuatro')
>>> a
>>> type(a)
<class 'NoneType'>
```

### 2.3. Añadir elementos a un diccionario en Python

Como hemos comentado, la clase `dict` es mutable, por lo que se pueden añadir, modificar y/o eliminar elementos después de haber creado un objeto de este tipo.

Para añadir un nuevo elemento a un diccionario existente, se usa el operador de asignación `=`. A la izquierda del operador aparece el objeto diccionario con la nueva clave entre corchetes `[]` y a la derecha el valor que se asocia a dicha clave.

```
>>> d = {'uno': 1, 'dos': 2}
>>> d
{'uno': 1, 'dos': 2}
# Añade un nuevo elemento al diccionario
>>> d['tres'] = 3
>>> d
{'uno': 1, 'dos': 2, 'tres': 3}
```

**NOTA:** ***Si la clave ya existe en el diccionario, se actualiza su valor.***

También existe el método `setdefault(clave[, valor])`. Este método devuelve el valor de la clave si ya existe y, en caso contrario, le asigna el valor que se pasa como segundo argumento. Si no se especifica este segundo argumento, por defecto es `None`.

```
>>> d = {'uno': 1, 'dos': 2}
>>> d.setdefault('uno', 1.0)
1
>>> d.setdefault('tres', 3)
3
>>> d.setdefault('cuatro')
>>> d
{'uno': 1, 'dos': 2, 'tres': 3, 'cuatro': None}
```

### 2.4. Modificar elementos de un diccionario

En el apartado anterior hemos visto que, para actualizar el valor asociado a una clave, simplemente se asigna un nuevo valor a dicha clave del diccionario.

```
>>> d = {'uno': 1, 'dos': 2}
>>> d
{'uno': 1, 'dos': 2}
>>> d['uno'] = 1.0
>>> d
{'uno': 1.0, 'dos': 2}
```

### 2.5. Eliminar un elemento de un diccionario en Python

En Python existen diversos modos de eliminar un elemento de un diccionario. Son los siguientes:

- `pop(clave [, valor por defecto])`: Si la `clave` está en el diccionario, elimina el elemento y devuelve su valor; si no, devuelve el `valor por defecto`. Si no se proporciona el `valor por defecto` y la `clave` no está en el diccionario, se lanza la excepción `KeyError`.
- `popitem()`: Elimina el último par *clave: valor* del diccionario y lo devuelve. Si el diccionario está vacío se lanza la excepción `KeyError`. (**NOTA:** En versiones anteriores a Python 3.7, se elimina/devuelve un par aleatorio, no se garantiza que sea el último).
- `del d[clave]`: Elimina el par *clave: valor*. Si no existe la clave, se lanza la excepción `KeyError`.
- `clear()`: Borra todos los pares *clave: valor* del diccionario.

```
>>> d = {'uno': 1, 'dos': 2, 'tres': 3, 'cuatro': 4, 'cinco': 5}

# Elimina un elemento con pop()
>>> d.pop('uno')
1
>>> d
{'dos': 2, 'tres': 3, 'cuatro': 4, 'cinco': 5}

# Trata de eliminar una clave con pop() que no existe
>>> d.pop(6)
Traceback (most recent call last):
File "<input>", line 1, in <module>
KeyError: 6

# Elimina un elemento con popitem()
>>> d.popitem()
('cinco', 5)
>>> d
{'dos': 2, 'tres': 3, 'cuatro': 4}

# Elimina un elemento con del
>>> del d['tres']
>>> d
{'dos': 2, 'cuatro': 4}

# Trata de eliminar una clave con del que no existe
>>> del d['seis']
Traceback (most recent call last):
File "<input>", line 1, in <module>
KeyError: 'seis'

# Borra todos los elementos del diccionario
>>> d.clear()
>>> d
{}
```

## 3. Patrones de uso de diccionarios en Python

Python es un lenguaje muy versátil y flexible que permite utilizar los diccionarios de muchas formas. A continuación, se describen algunos de los patrones de uso más comunes de los diccionarios en Python.

### 3.1. Número de elementos (len) de un diccionario en Python

Al igual que sucede con otros tipos contenedores, se puede usar la función de Python `len()` para obtener el número de elementos de un diccionario.

```
>>> d = {'uno': 1, 'dos': 2, 'tres': 3}
>>> len(d)
3
```

### 3.2. Recorrer un diccionario - for dict Python

Hay varias formas de recorrer los elementos de un diccionario: recorrer solo las claves, solo los valores o recorrer a la vez las claves y los valores. Puedes ver aquí cómo usar el bucle for para recorrer un diccionario.

```
>>> d = {'uno': 1, 'dos': 2, 'tres': 3}
>>> for e in d:
...     print(e)
...
uno
dos
tres

# Recorrer las claves del diccionario
>>> for k in d.keys():
...     print(k)
...
uno
dos
tres

# Recorrer los valores del diccionario
>>> for v in d.values():
...     print(v)
...
1
2
3

# Recorrer los pares clave valor
>>> for i in d.items():
...     print(i)
...
('uno', 1)
('dos', 2)
('tres', 3)
```

### 3.3. Comprobar si un elemento está en un diccionario en Python

Al operar con diccionarios, se puede usar el operador de pertenencia `in` para comprobar si una clave está contenida, o no, en un diccionario. Esto resulta útil, por ejemplo, para asegurarnos de que una clave existe antes de intentar eliminarla.

```
>>> print('uno' in d)
True
>>> print(1 in d)
False
>>> print(1 not in d)
True
# Intenta eliminar la clave 1 si existe
>>> if 1 in d:
...     del d[1]
...
>>> d
{'uno': 1, 'dos': 2, 'tres': 3}
```

### 3.4. Comparar si dos diccionarios son iguales

En Python se puede utilizar el operador de igualdad == para comparar si dos diccionarios son iguales. **Dos diccionarios son iguales si contienen el mismo conjunto de pares *clave: valor*** , independientemente del orden que tengan.

Otro tipo de comparaciones entre diccionarios no están permitidas. Si se intenta, el intérprete lanzará la excepción `TypeError`.

```
>>> d1 = {'uno': 1, 'dos': 2}
>>> d2 = {'dos': 2, 'uno': 1}
>>> d3 = {'uno': 1}
>>> print(d1 == d2)
True
>>> print(d1 == d3)
False
>>> print(d1 > d2)
Traceback (most recent call last):
File "<input>", line 1, in <module>
TypeError: '>' not supported between instances of 'dict' and 'dict'
```

### 3.5. Diccionarios anidados en Python

Un diccionario puede contener un valor de cualquier tipo, entre ellos, otro diccionario. Este hecho se conoce como diccionarios anidados.

Para acceder al valor de una de las claves de un diccionario interno, se usa el operador de indexación anidada `[clave1][clave2]...`

Veámoslo con un ejemplo:

```
>>> d = {'d1': {'k1': 1, 'k2': 2}, 'd2': {'k1': 3, 'k4': 4}}
>>> d['d1']['k1']
1
>>> d['d2']['k1']
3
```

```
>>> d['d2']['k4']
4
>>> d['d3']['k4']
Traceback (most recent call last):
File "<input>", line 1, in <module>
KeyError: 'd3'
```

### 3.6. Obtener una lista con las claves de un diccionario

En ocasiones, es necesario tener almacenado en una lista las claves de un diccionario. Para ello, simplemente pasa el diccionario como argumento del constructor `list()`. Esto devolverá las claves del diccionario en una lista.

```
>>> d = {'uno': 1, 'dos': 2, 'tres': 3}
>>> list(d)
['uno', 'dos', 'tres']
```

### 3.7. Objetos vista de un diccionario

La clase *dict* implementa tres métodos muy particulares, dado que devuelven un tipo de dato, *iterable*, conocido como *objetos vista*. Estos objetos ofrecen una vista de las claves y valores contenidos en el diccionario y si el diccionario se modifica, dichos objetos se actualizan al instante.

- `keys()`: Devuelve una vista de las claves del diccionario.
- `values()`: Devuelve una vista de los valores del diccionario.
- `items()`: Devuelve una vista de pares *(clave, valor)* del diccionario.

```
>>> d = {'uno': 1, 'dos': 2, 'tres': 3}
# d.keys() es diferente a list(d), aunque ambos
# contengan las claves del diccionario
# d.keys() es de tipo dict_keys y list(d) es de tipo list
>>> v = d.keys()
>>> type(v)
<class 'dict_keys'>
>>> v
dict_keys(['uno', 'dos', 'tres'])

>>> l = list(d)
>>> type(l)
<class 'list'>
>>> l
['uno', 'dos', 'tres']

>>> v = d.values()
>>> type(v)
<class 'dict_values'>
>>> v
dict_values([1, 2, 3])

>>> v = d.items()
>>> type(v)
<class 'dict_items'>
>>> v
dict_items([('uno', 1), ('dos', 2), ('tres', 3)])
```

## 4. Listado de métodos de la clase dict

Finalmente, enumeramos 0el listado de los principales métodos de la clase *dict*. Algunos de ellos ya los hemos visto durante la unidad:

| Método | Descripción |
|---|---|
| `clear()` | Elimina todos los elementos del diccionario. |
| `copy()` | Devuelve una copia poco profunda del diccionario. |
| `get(clave[, valor])` | Devuelve el valor de la`clave`. Si no existe, devuelve el valor `valor` si se indica y si no, `None`. |
| `items()` | Devuelve una vista de los pares*clave: valor* del diccionario. |
| `keys()` | Devuelve una vista de las claves del diccionario. |
| `pop(clave[, valor])` | Devuelve el valor del elemento cuya clave es`clave` y elimina el elemento del diccionario. Si la clave no se encuentra, devuelve `valor` si se proporciona. Si la clave no se encuentra y no se indica `valor`, lanza la excepción `KeyError`. |
| `popitem()` | Devuelve un par*(clave, valor)* aleatorio del diccionario. Si el diccionario está vacío, lanza la excepción`KeyError`. |
| `setdefault(clave[, valor])` | Si la`clave` está en el diccionario, devuelve su valor. Si no lo está, inserta la `clave` con el valor `valor` y lo devuelve (si no se especifica `valor`, por defecto es `None`). |
| `update(iterable)` | Actualiza el diccionario con los pares*clave: valor* del `iterable`. |
| `values()` | Devuelve una vista de los valores del diccionario. |

## 5. Depuración

Conforme trabajes con conjuntos de datos más grandes puede ser complicado depurar imprimiendo y revisando los datos a mano. Aquí hay algunas sugerencias para depurar grandes conjuntos de datos:

- Reducir la entrada: Si es posible, trata de reducir el tamaño del conjunto de datos. Por ejemplo, si el programa lee un archivo de texto, comienza solamente con las primeras 10 líneas, o con el ejemplo más pequeño que puedas encontrar. Puedes ya sea editar los archivos directamente, o (mejor) modificar el programa para que solamente lea las primeras `n` número de líneas.

Si hay un error, puedes reducir `n` al valor más pequeño que produce el error, y después incrementarlo gradualmente conforme vayas encontrando y corrigiendo errores.

- Revisar extractos y tipos: En lugar de imprimir y revisar el conjunto de datos completo, considera imprimir extractos de los datos: por ejemplo, el número de elementos en un diccionario o el total de una lista de números.

Una causa común de errores en tiempo de ejecución es un valor que no es el tipo correcto. Para depurar este tipo de error, generalmente es suficiente con imprimir el tipo de un valor.

- Escribe auto-verificaciones: Algunas veces puedes escribir código para revisar errores automáticamente. Por ejemplo, si estás calculando el promedio de una lista de números, podrías verificar que el resultado no sea más grande que el elemento más grande de la lista o que sea menor que el elemento más pequeño de la lista. Esto es llamado “prueba de sanidad” porque detecta resultados que son “completamente ilógicos”.

Otro tipo de prueba compara los resultados de dos diferentes cálculos para ver si son consistentes. Esto es conocido como “prueba de consistencia”.

Imprimir una salida ordenada: Dar un formato a los mensajes de depuración puede facilitar encontrar un error.

De nuevo, el tiempo que inviertas haciendo una buena estructura puede reducir el tiempo que inviertas en depurar.

# Fuente

- Pagina de Juan Jose Lozano Gomez sobre Python
- Estructuras de datos
- Python para todos
- Dicionarios y Hash
- Aprende con Alf

# * Práctica 3.2: Diccionarios

## P3.2 - Ejercicios: Diccionarios

# 3.4.-Est.Datos: Conjuntos

# 3.4. Conjuntos

Los conjuntos se han convertido en una estructura de datos muy utilizada en Python. En esta unidad vamos a ver qué es un conjunto, cómo se crea, cómo se añaden y eliminan elementos, cómo se accede a los elementos, cómo se realizan operaciones con conjuntos, etc.

## 1. Qué es el tipo set en Python

Los conjuntos son una estructura de datos que permite almacenar elementos de forma desordenada y únicos, es decir, sin repetición. En Python, los conjuntos, tipo `set`, se representan con llaves `{}` y los elementos se separan por comas. Los conjuntos son mutables, es decir, se pueden modificar después de su creación,

### 1.1. Características de los conjuntos en Python

La principal característica de este tipo de datos es que es una colección cuyos elementos *no guardan ningún orden* y que además *son únicos*.

Estas características hacen que los principales usos de esta clase sean conocer si un elemento pertenece o no a una colección y eliminar duplicados de un tipo secuencial (list, tuple o str).

Además, esta clase también implementa las típicas operaciones matemáticas sobre conjuntos: *unión* , *intersección* , *diferencia* , …

Los principales usos de los conjuntos son:

- Eliminar duplicados de una lista. Ej: `set([1, 2, 3, 1, 2])` devuelve `{1, 2, 3}`.
- Comprobar si un elemento está en una colección. Ej: `3 in {1, 2, 3}` devuelve `True`.
- Realizar operaciones de conjuntos. Ej: `{1, 2, 3} & {2, 3, 4}` devuelve `{2, 3}`.
- Implementar algoritmos de búsqueda y optimización. Ej: `set([1, 2, 3]) - set([2, 3, 4])` devuelve `{1}`.

## 2. Trabajar con conjuntos

A continuación, se muestra cómo trabajar con conjuntos en Python. Se verá cómo crear conjuntos, añadir y eliminar elementos, acceder a los elementos.

### 2.1. Creación de conjuntos en Python

Para crear un conjunto, basta con encerrar una serie de elementos entre llaves `{}`, o bien usar el constructor de la clase `set()` y pasarle como argumento un objeto *iterable* (como una *lista* , una *tupla* , una *cadena* …).

```
# Crea un conjunto con una serie de elementos entre llaves
# Los elementos repetidos se eliminan
>>> c = {1, 3, 2, 9, 3, 1}
>>> c
{1, 2, 3, 9}

# Crea un conjunto a partir de un string
# Los caracteres repetidos se eliminan
>>> a = set('Hola Pythonista')
>>> a
{'a', 'H', 'h', 'y', 'n', 's', 'P', 't', ' ', 'i', 'l', 'o'}

# Crea un conjunto a partir de una lista
# Los elementos repetidos de la lista se eliminan
>>> unicos = set([3, 5, 6, 1, 5])
>>> unicos
{1, 3, 5, 6}
```

Para crear un conjunto vacío, simplemente llama al constructor `set()` sin parámetros.

❗ **IMPORTANTE: ** `{}` NO crea un conjunto vacío, sino un *diccionario* vacío. Usa `set()` si quieres crear un conjunto sin elementos.

**NOTA**: **Los elementos que se pueden añadir a un conjunto deben ser de tipo *hashable*. Un objeto es *hashable* si tiene un valor de *hash* que no cambia durante todo su ciclo de vida. En principio, los objetos que son instancias de clases definidas por el usuario son *hashables*. También lo son la mayoría de tipos inmutables definidos por Python.**

#### 2.1.1. set vs frozenset

En realidad, en Python podemos crear dos clases de conjuntos: `set` y `frozenset`. La principal diferencia es que `set` es mutable, por lo que después de ser creado, se pueden añadir y/o eliminar elementos del conjunto, como veremos en secciones posteriores. Por su parte, `frozenset` es inmutable y su contenido no puede ser modificado una vez que ha sido inicializado.

Para crear un conjunto de tipo `frozenset`, se usa el constructor de la clase `frozenset()`:

```
>>> f = frozenset([3, 5, 6, 1, 5])
>>> f
frozenset({1, 3, 5, 6})
```

**NOTA:** El único modo en Python de tener un conjunto de conjuntos es utilizando objetos de tipo *frozenset* como elementos del propio conjunto.

### 2.2. Cómo acceder a los elementos de un conjunto en Python

Dado que los conjuntos son colecciones desordenadas, en ellos no se guarda la posición en la que son insertados los elementos como ocurre en los tipos `list` o `tuple`. Es por ello que **no se puede acceder a los elementos a través de un índice**.

Sin embargo, sí se puede acceder y/o recorrer todos los elementos de un conjunto usando un bucle for:

```
>>> mi_conjunto = {1, 3, 2, 9, 3, 1}
>>> for e in mi_conjunto:
...     print(e)
...
1
2
3
9
```

### 2.3. Añadir elementos a un conjunto (set) en Python

Para añadir un elemento a un conjunto se utiliza el método `add()`. También existe el método `update()`, que puede tomar como argumento una *lista*, *tupla*, *string*, *conjunto* o cualquier objeto de tipo *iterable*.

```
>>> mi_conjunto = {1, 3, 2, 9, 3, 1}
>>> mi_conjunto
{1, 2, 3, 9}

# Añade el elemento 7 al conjunto
>>> mi_conjunto.add(7)
>>> mi_conjunto
{1, 2, 3, 7, 9}

# Añade los elementos 5, 3, 4 y 6 al conjunto
# Los elementos repetidos no se añaden al conjunto
>>> mi_conjunto.update([5, 3, 4, 6])
>>> mi_conjunto
{1, 2, 3, 4, 5, 6, 7, 9}
```

**NOTA:** `add()` y `update()` no añaden elementos que ya existen al conjunto.

### 2.4. Eliminar un elemento de un conjunto en Python

La clase `set` ofrece cuatro métodos para eliminar elementos de un conjunto. Son: `discard()`, `remove()`, `pop()` y `clear()`. A continuación, se explica qué hace cada uno de ellos.

- `discard(elemento)` y `remove(elemento)` eliminan `elemento` del conjunto. La única diferencia es que si `elemento` no existe, `discard()` no hace nada mientras que **`remove()` lanza la excepción `KeyError`.**
- `pop()` es un tanto peculiar. Este método **devuelve un elemento aleatorio** del conjunto y lo elimina del mismo. Si el conjunto está vacío, lanza la excepción `KeyError`.
- `clear()` elimina todos los elementos contenidos en el conjunto.

```
>>> mi_conjunto = {1, 3, 2, 9, 3, 1, 6, 4, 5}
>>> mi_conjunto
{1, 2, 3, 4, 5, 6, 9}

# Elimina el elemento 1 con remove()
>>> mi_conjunto.remove(1)
```

```
>>> mi_conjunto
{2, 3, 4, 5, 6, 9}

# Elimina el elemento 4 con discard()
>>> mi_conjunto.discard(4)
>>> mi_conjunto
{2, 3, 5, 6, 9}

# Trata de eliminar el elemento 7 (no existe) con remove()
# Lanza la excepción KeyError
>>> mi_conjunto.remove(7)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
KeyError: 7

# Trata de eliminar el elemento 7 (no existe) con discard()
# No hace nada
>>> mi_conjunto.discard(7)
>>> mi_conjunto
{2, 3, 5, 6, 9}

# Obtiene y elimina un elemento aleatorio con pop()
>>> mi_conjunto.pop()
2
>>> mi_conjunto
{3, 5, 6, 9}

# Elimina todos los elementos del conjunto
>>> mi_conjunto.clear()
>>> mi_conjunto
set()
```

## 3. Patrones de uso de conjuntos en Python

Veamos algunos ejemplos de uso de conjuntos en Python.

### 3.1. Número de elementos (len) de un conjunto

Como con cualquier otra colección, puedes usar la función `len()` para obtener el número de elementos contenidos en un conjunto:

```
>>> mi_conjunto = set([1, 2, 5, 3, 1, 5])
>>> len(mi_conjunto)
4
```

### 3.2. Cómo saber si un elemento está en un conjunto

Con los conjuntos también se puede usar el operador de pertenencia `in` para comprobar si un elemento está contenido, o no, en un conjunto:

```
>>> mi_conjunto = set([1, 2, 5, 3, 1, 5])
>>> print(1 in mi_conjunto)
True

>>> print(6 in mi_conjunto)
False
```

```
>>> print(2 not in mi_conjunto)
False
```

## 3.3. Operaciones sobre conjuntos en Python (set operations)

Uno de los principales usos del tipo `set` es utilizarlo en operaciones del álgebra de conjuntos: *unión*, *intersección*, *diferencia*, *diferencia simétrica*, …

A continuación, veremos cómo llevar a cabo estas operaciones en Python.

### 3.3.1. Unión de conjuntos en Python

La unión de dos conjuntos `A` y `B` es el conjunto `A ∪ B` que contiene todos los elementos de `A` y de `B`.

En Python se utiliza el operador `|` para realizar la unión de dos o más conjuntos.

```
>>> a = {1, 2, 3, 4}
>>> b = {2, 4, 6, 8}
>>> a | b
{1, 2, 3, 4, 6, 8}
```

### 3.3.2. Intersección de conjuntos en Python

La intersección de dos conjuntos `A` y `B` es el conjunto `A ∩ B` que contiene todos los elementos comunes de `A` y de `B`.

En Python se utiliza el operador `&` para realizar la intersección de dos o más conjuntos.

```
>>> a = {1, 2, 3, 4}
>>> b = {2, 4, 6, 8}
>>> a & b
{2, 4}
```

### 3.3.3. Diferencia de conjuntos en Python

La diferencia entre dos conjuntos `A` y `B` es el conjunto `A \ B` que contiene todos los elementos de `A` que no pertenecen a `B`.

En Python se utiliza el operador `-` para realizar la diferencia de dos o más conjuntos.

```
>>> a = {1, 2, 3, 4}
>>> b = {2, 4, 6, 8}
>>> a - b
{1, 3}
```

### 3.3.4. Diferencia simétrica de conjuntos en Python

La diferencia simétrica entre dos conjuntos `A` y `B` es el conjunto que contiene los elementos de `A` y `B` que no son comunes.

En Python se utiliza el operador `^` para realizar la diferencia simétrica de dos o más conjuntos.

```
>>> a = {1, 2, 3, 4}
```

```
>>> b = {2, 4, 6, 8}
a ^ b
{1, 3, 6, 8}
```

### 3.3.4. Inclusión de conjuntos en Python

Dado un conjunto `A`, subcolección del conjunto `B` o igual a este, sus elementos son un subconjunto de `B`. Es decir, `A` es un subconjunto de `B` y `B` es un superconjunto de `A`.

En Python se utiliza el operador `<=` para comprobar si un conjunto `A` es subconjunto de `B` y el operador `>=` para comprobar si un conjunto `A` es superconjunto de `B`.

```
>>> a = {1, 2}
>>> b = {1, 2, 3, 4}
>>> a <= b
True

>>> a >= b
False

>>> b >= a
True

>>> a = {1, 2}
>>> b = {1, 2}
>>> a < b  # Ojo al operador < sin el =
False

>>> a <= b
True
```

### 3.3.5. Conjuntos disjuntos en Python

Dos conjuntos `A` y `B` son disjuntos si no tienen elementos en común, es decir, la intersección de `A` y `B` es el conjunto vacío.

En Python se utiliza el método `isdisjoint()` de la clase `set` para comprobar si un conjunto es disjunto de otro.

```
>>> a = {1, 2}
>>> b = {1, 2, 3, 4}
>>> a.isdisjoint(b)
False

>>> a = {1, 2}
>>> b = {3, 4}
>>> a.isdisjoint(b)
True
```

### 3.3.6. Igualdad de conjuntos en Python

En Python dos conjuntos son iguales si y solo si todos los elementos de un conjunto están contenidos en el otro. Esto quiere decir que cada uno es un subconjunto del otro.

```
>>> a = {1, 2}
>>> b = {1, 2}
>>> id(a)
```

```
4475070656
>>> id(b)
4475072096
>>> a == b
True
```

## 4. Métodos de la clase set en Python

Se finaliza, listando los métodos principales de la clase `set` en Python:

| Método | Descripción |
|---|---|
| `add(e)` | Añade un elemento al conjunto. |
| `clear()` | Elimina todos los elementos del conjunto. |
| `copy()` | Devuelve una copia superficial del conjunto. |
| `difference(iterable)` | Devuelve la diferencia del conjunto con el`iterable` como un conjunto nuevo. |
| `difference_update(iterable)` | Actualiza el conjunto tras realizar la diferencia con el`iterable`. |
| `discard(e)` | Elimina, si existe, el elemento del conjunto. |
| `intersection(iterable)` | Devuelve la intersección del conjunto con el`iterable` como un conjunto nuevo. |
| `intersection_update(iterable)` | Actualiza el conjunto tras realizar la intersección con el`iterable`. |
| `isdisjoint(iterable)` | Devuelve`True` si dos conjuntos son disjuntos. |
| `issubset(iterable)` | Devuelve`True` si el conjunto es subconjunto del `iterable`. |
| `issuperset(iterable)` | Devuelve`True` si el conjunto es superconjunto del `iterable`. |
| `pop()` | Obtiene y elimina un elemento de forma aleatoria del conjunto. |
| `remove(e)` | Elimina el elemento del conjunto. Si no existe lanza un error. |
| `symmetric_difference(iterable)` | Devuelve la diferencia simétrica del conjunto con el`iterable` como un conjunto nuevo. |

| Método | Descripción |
| --- | --- |
| `symmetric_difference_update(iterable)` | Actualiza el conjunto tras realizar la diferencia simétrica con el`iterable`. |
| `union(iterable)` | Devuelve la unión del conjunto con el`iterable` como un conjunto nuevo. |
| `update(iterable)` | Actualiza el conjunto tras realizar la unión con el`iterable`. |

**NOTA: Los operadores `|`, `&`, … toman siempre como operandos objetos de tipo `set`. Sin embargo, sus respectivas versiones como métodos `union()`, `intersection()`, … toman como argumentos un *iterable* (*lista, tupla, conjunto*, etc.).**

# Fuente

- Página de Juan Jose Lozano Gomez sobre Python
- Estructuras de datos
- Python para todos
- Aprende con Alf
- Operaciones con conjuntos

# * Práctica 3.3: Conjuntos