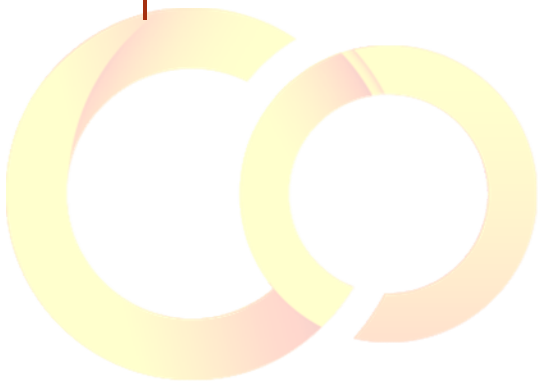


Unidad 6

Colecciones y Consultas

Manual del Estudiante



CLOUD
FORMACIÓN

Maria del Carmen Sierra Fernández

Contenido

Nivel II: Representación, proceso y visualización de Datos.....	2
Unidad 6: Colecciones y Consultas	2
Listas.....	2
Métodos de las Listas.....	4
Usar listas como pilas.....	9
Usando listas como colas	10
Tuplas.....	11
Conjuntos.....	13
Diccionarios.....	14
Operaciones de diccionario	16
Métodos.....	16
Alias y copia.....	18
Acumulación de varios resultados en un diccionario	19
Cuando usar un diccionario	21
Ordenar Diccionarios	22
Función Sorted ().....	22
Ordenar colecciones por más de una propiedad.....	25
Ordenar una tupla.....	25
Ordenar una lista	26
Ordenar un diccionario	26
Consultas.....	33
Funciones	34
Otras funciones para objetos iterables.....	34
modulo Fn.py	40
modulo cytoolz.....	41
Funciones cytoolz.....	41
Ordenando objetos con operator.....	42

Nivel II: Representación, proceso y visualización de Datos

Unidad 6: Colecciones y Consultas

Además de los tipos de datos vistos en la Unidad 1, Python cuenta con tipos más complejos que denominamos colecciones, porque nos permiten agrupar elementos, ellos son las listas, tuplas y los diccionarios.

Podemos decir, que una colección permite agrupar varios objetos bajo un mismo nombre.

Listas

Una lista es una secuencia mutable, utilizada para almacenar información. Las listas en Python nos permiten almacenar cualquier tipo de valor como enteros, cadenas y hasta otras funciones, para acceder a estos datos podemos hacer mediante un índice.

Las listas se pueden construir de varias maneras:

- Utilizando un par de corchetes para denotar la lista vacía: `[]`
- Utilizando corchetes, separando los elementos con comas: `[a]`, `[a, b, c]`
- Utilizando una comprensión de lista: `[x for x in iterable]`
- Utilizando el constructor de tipo: `list()` o `list(iterable)`

Veamos algunos ejemplos:

```
# Utilizando un par de corchetes para denotar la lista vacía: []
```

```
mi_lista = []  
mi_lista  
[]
```

```
# Utilizando corchetes, separando elementos con comas: [a], [a,b,c]
```

```
letras = ["a", "b", "c"]  
letras  
['a', 'b', 'c']
```

```
letra = ["a"]  
letra  
['a']
```

```
# Utilizando una comprensión de lista: [x para x en iterable]
```

```
lista = [ i for i in range(0,4)]  
lista  
[0, 1, 2, 3]
```

Nivel II: Representación, proceso y visualización de Datos

```
cuadrados = list(map(lambda x: x**2, range(10)))
cuadrados
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Utilizando el constructor de tipo: `list()` o `list(iterable)`

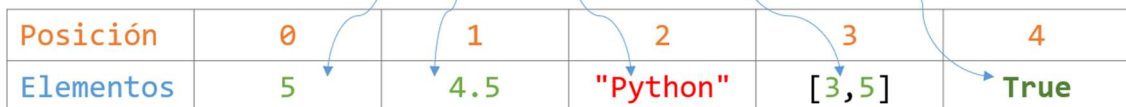
```
list('abc')
['a', 'b', 'c']
```

```
list(range(0,11))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
list((1, 2, 3))
[1, 2, 3]
```

Una lista es un arreglo de elementos donde podemos ingresar cualquier tipo de dato, para acceder a estos datos podemos hacer mediante un índice, que corresponde a la posición del elemento dentro de la lista.

```
lista = [5, 4.5, "Python", [3,5], True]
```



Posición	0	1	2	3	4
Elementos	5	4.5	"Python"	[3,5]	True

Veamos ahora como podemos acceder a cada elemento de la lista:

```
lista[0]
5
```

```
lista[1]
4.5
```

```
lista[2]
'Python'
```

```
lista[3]
[3, 5]
```

El elemento 3 de nuestra lista, es una lista [3, 5], para acceder a cada elemento debemos:

```
lista[3][0]
3
```

```
lista[3][1]
5
```

```
lista[4]  
True
```

También podemos acceder a varios elementos `list[start:end:step]`:

```
lista[1:3] # Extrae los elementos 1 y 2  
[4.5, 'Python']  
  
lista[:3] # Extrae los elementos de las posiciones 0,1 y 2 (< 3)  
[5, 4.5, 'Python']  
  
lista[3:] # Extrae los elementos de las posiciones 3 y 4 (> 3)  
[[3, 5], True]  
  
lista[0:3:2] # Extrae los elementos de las posiciones 0,1 y 2  
con un salto de 2  
[5, 'Python']
```

También podemos obtener la longitud o total de elementos de la lista con `len()`

```
len(lista)  
5
```

Las listas son iterables, así que podemos recorrer cada elemento de la lista utilizando un bucle `for`:

```
for l in lista:  
    print(l)  
5  
4.5  
Python  
[3, 5]  
True
```

Métodos de las Listas

Las listas en Python tienen muchos métodos que podemos utilizar, ahora vamos a nombrar los más importantes:

`append(x)`: Este método nos permite agregar nuevos elementos al final de una lista.

```
lista.append(10)  
lista  
[5, 4.5, 'Python', [3, 5], True, 10]  
  
lista.append([1,2,3])  
lista
```

```
[5, 4.5, 'Python', [3, 5], True, 10, [1, 2, 3]]
```

Podemos agregar cualquier tipo de elemento a una lista, pero tengan en cuenta lo que pasa cuando agregamos una lista dentro de otra, esta lista se agrega como uno y solo un elemento.

`extend(iterable)`: Extiende la lista agregándole todos los ítems del iterable. `extend` también nos permite agregar elementos dentro de una lista, pero a diferencia de `append` al momento de agregar una lista, cada elemento de esta lista se agrega como un elemento más dentro de la otra lista.

```
lista.extend([2,5])
lista
[5, 4.5, 'Python', [3, 5], True, 10, [1, 2, 3], 2, 5]
```

`insert(i, x)`: Inserta un ítem en una posición dada. El primer argumento es el índice del ítem delante del cual se insertará, por lo tanto `a.insert(0, x)` inserta al principio de la lista y `a.insert(len(a), x)` equivale a `a.append(x)`.

```
lista.insert(0,"insertado")
lista
['insertado', 5, 4.5, 'Python', [3, 5], True, 10, [1, 2, 3], 2, 5]
```

`remove(x)`: Quita el primer ítem de la lista cuyo valor sea `x`. Lanza un `ValueError` si no existe tal ítem. En este ejemplo, removeremos el elemento 2, de la lista :

```
lista.remove(2) # elimina el elemento con valor 2 de lista
lista
['insertado', 5, 4.5, 'Python', [3, 5], True, 10, [1, 2, 3], 5]
```

`pop([i])`: Quita el ítem en la posición dada de la lista y lo retorna. Si no se especifica un índice, `lista.pop()` quita y retorna el último elemento de la lista.

```
valor = lista.pop(2)
valor
4.5

lista
['insertado', 5, 'Python', [3, 5], True, 10, [1, 2, 3], 5]
```

`del()`: Hay una manera de quitar un ítem de una lista dado su índice en lugar de su valor: la instrucción `del`. Esta es diferente del método `pop()`, el cual retorna un valor. La instrucción `del` también puede usarse para quitar secciones de una lista o vaciar la lista completa. Por ejemplo:

```
lista = ['insertado', 5, 'Python', [3, 5], True, 10, [1, 2, 3], 5]
```

Nivel II: Representación, proceso y visualización de Datos

```
del lista[0] # Eliminamos el elemento de la posición 0
[5, 'Python', [3, 5], True, 10, [1, 2, 3], 5]

del lista[2:4] # Eliminamos los elementos de la posición 2 y 3
lista
[5, 'Python', 10, [1, 2, 3], 5]

del lista[:] # Eliminamos todos los elementos de la lista
lista
[]
```

`index(x[, start[, end]])`: Retorna el índice basado en cero del primer elemento cuyo valor sea igual a x.

Los argumentos opcionales `start` y `end` se utilizan para limitar la búsqueda a un segmento particular de la lista. El índice retornado se calcula de manera relativa al inicio de la secuencia completa en lugar de con respecto al argumento `start`.

```
lista.index("Python") # retorna la posición del elemento
"Python"
2
```

`index()` lanza una excepción `ValueError` si no existe tal elemento:

```
lista.index("Py")
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-34-a8c93f0464ca> in <module>
----> 1 lista.index("Py")

ValueError: 'Py' is not in list
```

Podemos gestionar la excepción, para evitar la salida del programa

```
try:
    lista.index("Py")
except:
    print("No encontró el elemento")
```

`count(x)`: Retorna el número de veces que x aparece en la lista.

```
lista.count(5)
2
```

En el ejemplo, contamos cuantas veces se repite el número 5 dentro de la lista, y esto retorna 2.

Nivel II: Representación, proceso y visualización de Datos

`sort(key=None, reverse=False)`: Ordena los elementos de la lista en su lugar. `sort()` acepta dos argumentos que solo se pueden pasar por palabra clave (argumentos de solo palabra clave):

- **key**: especifica una función de un argumento que se utiliza para extraer una clave de comparación de cada elemento de la lista (por ejemplo, `key = str.lower`). La clave correspondiente a cada elemento de la lista se calcula una vez y luego se utiliza para todo el proceso de clasificación. El valor predeterminado es `None` significa que los elementos de la lista se ordenan directamente sin calcular un valor de clave independiente.
- **Reverse**: es un valor booleano. Si se establece en `True`, los elementos de la lista se ordenan como si cada comparación estuviera invertida.

El criterio de ordenación viene dado por los propios elementos. Si la lista contiene cadenas se ordena por orden alfabético. Si la lista contiene elementos de tipos diferentes se puede producir un error de ordenación, si son tipos incompatibles. Por ejemplo:

```
lista = [5, 4.5, 'Python', [3, 5], True]
lista.sort()
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-46-2bac7e0ec028> in <module>
----> 1 lista.sort()

TypeError: '<' not supported between instances of 'str' and 'float'
```

Podemos ordenar listas que contengas números enteros y números con punto flotante, pero no podemos ordenar listas que contengas números y cadenas.

```
numeros = [1, 8, 6.5, 9, 15.55, 2]
numeros.sort()
numeros
[1, 2, 6.5, 8, 9, 15.55]
```

Veamos otros ejemplos:

```
frutas = ['Manzana', 'Pera', 'Melón', 'Cerezas', 'Kiwi']
frutas
['Manzana', 'Pera', 'Melón', 'Cerezas', 'Kiwi']

# Ordenamos la lista frutas
frutas.sort()
frutas
['Cerezas', 'Kiwi', 'Manzana', 'Melón', 'Pera']
```

```
# Ordenamos la lista frutas en orden inverso
```


Nivel II: Representación, proceso y visualización de Datos

```
frutas.sort(reverse=True)
frutas
['Pera', 'Melón', 'Manzana', 'Kiwi', 'Cerezas']

# Insertamos kiwi en minúscula
frutas.append('kiwi')
frutas
['Pera', 'Melón', 'Manzana', 'Kiwi', 'Cerezas', 'kiwi']

# Ordenamos la lista con kiwi en minúscula
frutas.sort()
frutas
['Cerezas', 'Kiwi', 'Manzana', 'Melón', 'Pera', 'kiwi']
```

Observamos que “kiwi” en minúscula los coloca al final, para corregirlo vamos a ordenar la lista, pasando como parámetro `key` la función convertir en minúsculas:

```
frutas.sort(key = str.lower)
frutas
['Cerezas', 'Kiwi', 'kiwi', 'Manzana', 'Melón', 'Pera']
```

A `sort()`, también podemos pasarle como parámetro `key` nuestra función `calcular_longitud`, utilizando esta función ordenará nuestra lista por el tamaño o longitud de cada elemento:

```
def calcular_longitud(e):
    return len(e) # retorna la longitud de cada elemento

frutas.sort(key=calcular_longitud)
frutas
['Kiwi', 'kiwi', 'Pera', 'Melón', 'Cerezas', 'Manzana']
```

`reverse()`: Invierte los elementos de la lista en su lugar o in situ.

```
lista = ['a', 'b', 'c', 'f', 'd', 'h', 'i']
lista
['a', 'b', 'c', 'f', 'd', 'h', 'i']

lista.reverse()
lista
['i', 'h', 'd', 'f', 'c', 'b', 'a']
```

`list.copy()`: Retorna una copia superficial de la lista. Equivalente a `x[:]`.

```
lista = ['a', 'b', 'c', 'f', 'd', 'h', 'i']
nueva_lista = lista.copy()
nueva_lista
['a', 'b', 'c', 'f', 'd', 'h', 'i']
```

```

numeros = [1, 2, 3, 4, 5]
copia = numeros[:]
copia
[1, 2, 3, 4, 5]

```

Usar listas como pilas

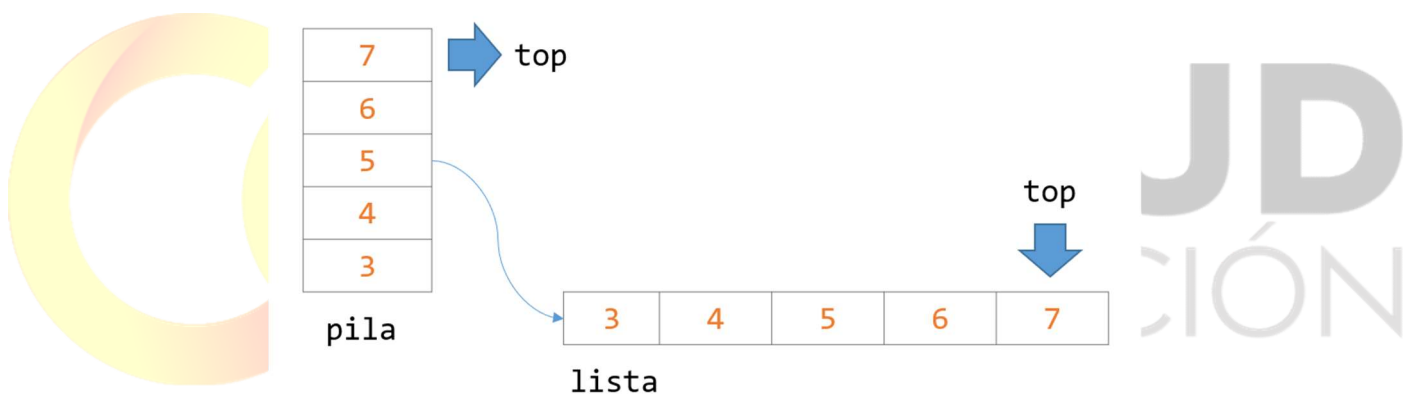
Los métodos de lista hacen que sea muy fácil usar una lista como una pila, donde el último elemento agregado es el primer elemento recuperado ("último en entrar, primero en salir"). Para agregar un elemento a la parte superior de la pila, use `append()`. Para recuperar un elemento de la parte superior de la pila, use `pop()` sin un índice explícito.

Por ejemplo:

```

pila = [3, 4, 5]
pila.append(6)
pila.append(7)
pila

```



Hemos creado nuestra pila, ahora saquemos cada elemento de la pila:

```

pila.pop()
7
pila.pop()
6
pila.pop()
5
pila.pop()
4
pila.pop()
3
pila
[]

```

Si ejecutamos `pila.pop()` y no hay elementos en la lista, genera un `IndexError`:

```

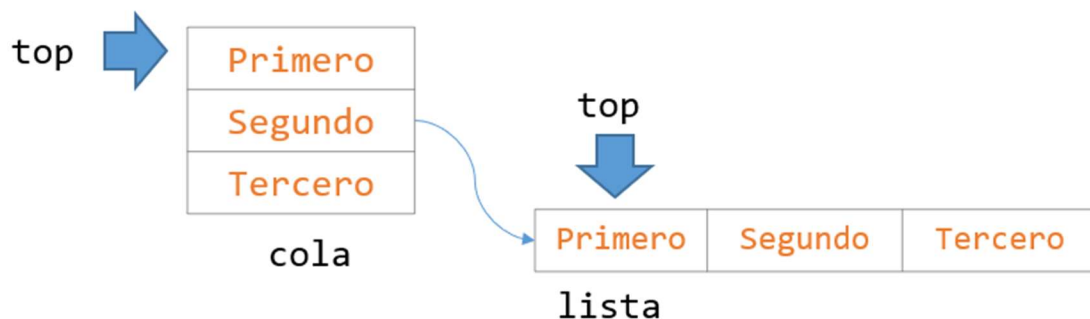
IndexError                                Traceback (most recent call last)
<ipython-input-116-73c5ba2ac2af> in <module>
----> 1 pila.pop()

```

IndexError: pop from empty list

Usando listas como colas

Es posible usar una lista como una cola, donde el primer elemento añadido es el primer elemento retirado («primero en entrar, primero en salir»); sin embargo, las listas no son eficientes para este propósito. Agregar y sacar del final de la lista es rápido, pero insertar o sacar del comienzo de una lista es lento (porque todos los otros elementos tienen que ser desplazados por uno).



Para implementar una cola, utiliza `collections.deque` el cual fue diseñado para añadir y quitar de ambas puntas de forma rápida. Por ejemplo:

```

from collections import deque
cola_clientes = deque(["Primero", "Segundo", "Tercero"])
cola_clientes
deque(['Primero', 'Segundo', 'Tercero'])

cola_clientes.append("Cuarto")    # Llega el cuarto cliente
cola_clientes.append("Quinto")   # Llega el quinto cliente
cola_clientes
deque(['Primero', 'Segundo', 'Tercero', 'Cuarto', 'Quinto'])

cola_clientes.popleft()          # Atiende al primer cliente
'Primero'

cola_clientes.popleft()          # Atiende al segundo cliente
'Segundo'

cola_clientes
deque(['Tercero', 'Cuarto', 'Quinto']) # Clientes en la cola

```

Otros métodos de la `collections.deque`: `appendleft()`

Listas anidadas

En Python podemos crear una lista de varias dimensiones, lo que se conoce en otros lenguajes como arreglo (arrays) o matrices.

Ejemplo: Tenemos las notas de tres grupos de estudiantes de tercero de la ESO y necesitamos calcular la media de los tres grupos:

```
# Notas de los alumnos 3ESO grupo A, B y C

notas_3ESO = [[5,3.5,6,8,3.5,10],
              [9,7.5,8,6.5,4,5],
              [10,4,5.5,8.3,7.5,9]]

# Obtener media de 3 ESO

sumatoria = 0
media = 0

for notas_grupo in notas_3ESO:
    for nota in notas_grupo:
        sumatoria = sumatoria + nota

contador = len(notas_3ESO[0]) + len(notas_3ESO[1]) +
           len(notas_3ESO[2])

media = sumatoria / contador

print("Media de 3 ESO: ", round(media,2))
Media de 3 ESO: 6.68
```

Tuplas

Una tupla es otro dato de tipo secuencia, consiste en un número de valores separados por comas, por ejemplo:

```
t = 456, 75.56, 'Python'
t
(456, 75.56, 'Python')

t = ("a", "b", "c")
t
('a', 'b', 'c')
```

Observamos en la salida las tuplas siempre se encierran entre paréntesis, para que las tuplas anidadas puedan interpretarse correctamente; pueden ingresarse con o sin paréntesis, aunque a menudo los paréntesis son necesarios de todas formas (si la tupla es parte de una

Nivel II: Representación, proceso y visualización de Datos

expresión más grande). No es posible asignar a los ítems individuales de una tupla, sin embargo, sí se puede crear tuplas que contengan objetos mutables, como las listas.

```
t[0] = "d"
```

```
-----  
TypeError                                Traceback (most recent call last)  
<ipython-input-51-3d223fe93c40> in <module>  
----> 1 t[0] = "d"  
  
TypeError: 'tuple' object does not support item assignment
```

```
v = ([1, 2, 3], [3, 2, 1]) # Pueden contener objetos mutables  
v  
([1, 2, 3], [3, 2, 1])
```

A pesar del parecido con las listas, las tuplas frecuentemente se utilizan en distintas situaciones y para distintos propósitos. Las tuplas son **inmutable** y normalmente contienen una secuencia heterogénea de elementos que son accedidos al desempaquetar o acceder por atributo. Las listas son mutables, y sus elementos son normalmente homogéneos y se acceden iterando a la lista.

Un problema particular es la construcción de tuplas que contengan 0 o 1 ítem: la sintaxis presenta algunas peculiaridades para estos casos. Las tuplas vacías se construyen mediante un par de paréntesis vacío; una tupla con un ítem se construye poniendo una coma a continuación del valor (no alcanza con encerrar un único valor entre paréntesis).

```
vacía = ()  
vacía  
()
```

```
uno = (1,)  
uno  
(1,)
```

La declaración

```
t = 456, 75.56, 'Python'
```

es un ejemplo de empaquetado de tuplas: los valores 456, 75.56 y 'Python' se empaquetan juntos en una tupla. La operación inversa también es posible:

```
x, y, z = t
```

```
x  
456  
y  
75.56
```

z
'Python'

Esto se llama, **desempaquetado de secuencias**, y funciona para cualquier secuencia en el lado derecho del igual. El desempaquetado de secuencias requiere que la cantidad de variables a la izquierda del signo igual sea el tamaño de la secuencia.

Conjuntos

Python también incluye un tipo de dato para conjuntos. Un conjunto es una colección no ordenada y sin elementos repetidos. Los usos básicos de éstos incluyen verificación de pertenencia y eliminación de entradas duplicadas.

```
frutas = {"manzana", "pera", "mandarina", "naranja", "manzana"}
frutas
{'mandarina', 'manzana', 'naranja', 'pera'}

# podemos validar si un elemento existe

"manzana" in frutas
True
"cereza" in frutas
False

# Son iterables

for fruta in frutas:
    print(fruta)

pera
manzana
naranja
mandarina
```

Las llaves o la función `set()` pueden usarse para crear conjuntos. Para crear un conjunto vacío tenemos que utilizar `set()`, no utilizar `{}`.

```
a = set('abracadabra')
a
{'a', 'b', 'c', 'd', 'r'}

b = set('alacazam')
b
{'a', 'c', 'l', 'm', 'z'}
```

Los conjuntos también soportan operaciones matemáticas como la unión, intersección, diferencia, y diferencia simétrica.

Operación	Operador	Método
Intersección	&	intersection()
Diferencia	-	difference()
Unión		union()
Unión exclusiva.	^	symmetric_difference()

```
a - b    # letras en a pero no en b
{'b', 'd', 'r'}
```

```
a.difference(b)
{'b', 'd', 'r'}
```

```
a | b    # letras en a o b, o en ambas
{'a', 'b', 'c', 'd', 'l', 'm', 'r', 'z'}
```

```
a.union(b)
{'a', 'b', 'c', 'd', 'l', 'm', 'r', 'z'}
```

```
a & b    # letras en ambas
{'a', 'c'}
```

```
a.intersection(b)
{'a', 'c'}
```

```
a ^ b    # letras en a o b, pero no en ambas
{'b', 'd', 'l', 'm', 'r', 'z'}
```

```
a.symmetric_difference(b)
{'b', 'd', 'l', 'm', 'r', 'z'}
```

De forma similar a las comprensiones de listas, está también soportada la comprensión de conjuntos:

```
z = {x for x in 'abracadabra' if x not in 'abc'}
z
{'d', 'r'}
```

Diccionarios

Es un objeto mutable de tipo mapa (mapping). A diferencia de las secuencias, que se indexan mediante un rango numérico, los diccionarios se indexan con claves, que pueden ser cualquier tipo inmutable; las cadenas y números siempre pueden ser claves. Las tuplas pueden usarse como claves si solamente contienen cadenas, números o tuplas; si una tupla contiene cualquier objeto mutable directa o indirectamente, no puede usarse como clave. No podemos utilizar listas como claves, ya que las listas pueden modificarse usando asignación por índice, asignación por sección, o métodos como `append()` y `extend()`.

Nivel II: Representación, proceso y visualización de Datos

Un diccionario se puede definir como un conjunto de pares `clave:valor` con el requerimiento de que las claves sean únicas (dentro de un diccionario).

Un par de llaves crean un diccionario vacío: `{}` o con la función `dict()`.

```
eng_esp = {}  
eng_esp  
{}  
  
a = dict()  
a  
{}
```

Colocar una lista de pares `clave:valor` separada por comas dentro de las llaves agrega, de inicio, pares `clave:valor` al diccionario.

```
eng_esp = {'three': 'tres', 'one': 'uno', 'two': 'dos'}  
eng_esp  
{'three': 'tres', 'one': 'uno', 'two': 'dos'}
```

Las siguientes asignaciones agregan nuevos pares clave-valor al diccionario. El lado izquierdo muestra el diccionario y la clave asociada. El lado derecho indica el valor asociado con esa clave.

```
eng_esp['five'] = "cinco"  
eng_esp['six'] = "seis"  
eng_esp  
{'three': 'tres', 'one': 'uno', 'two': 'dos', 'five': 'cinco',  
'six': 'seis'}
```

Los pares clave-valor del diccionario están separados por comas. Cada par contiene una clave y un valor separados por dos puntos.

Es posible que el orden de los pares no sea el esperado. Python utiliza algoritmos complejos, diseñados para un acceso muy rápido, para determinar dónde se almacenan los pares clave-valor en un diccionario. Para nuestros propósitos, podemos pensar en este orden como impredecible.

No importa en qué orden escribamos los pares. Se accede a los valores de un diccionario con claves, no con índices. Así es como usamos una clave para buscar el valor correspondiente.

```
value = eng_esp['two']  
print(value)  
dos  
  
print(eng_esp['one'])  
uno
```


Operaciones de diccionario

del: Elimina un par clave-valor de un diccionario.

```
inventario = {"manzanas":230, "peras":150, "naranjas":200}
inventario
{'manzanas': 230, 'peras': 150, 'naranjas': 200}

del inventario["peras"]
inventario
{'manzanas': 230, 'naranjas': 200}
```

Los diccionarios son mutables, esto significa que el diccionario se puede modificar haciendo referencia a una asociación en el lado izquierdo de la declaración de asignación. En el ejemplo anterior, en lugar de eliminar la entrada para las peras, podríamos haber establecido el inventario en 0.

```
inventario["peras"] = 0
inventario
{'manzanas': 230, 'naranjas': 200, 'peras': 0}
```

Ahora, imaginemos que hacemos una compra de 100 naranjas:

```
inventario["naranjas"] = inventario["naranjas"] + 100
inventario
{'manzanas': 230, 'naranjas': 300, 'peras': 0}
```

Ahora hay 300 naranjas; el diccionario ha sido modificado. La función `len` también funciona en diccionarios. Devuelve el número de pares clave-valor.

```
items = len(inventario)
items
3
```

Métodos

Los diccionarios tienen varios métodos incorporados.

Método	Parámetros	Descripción
keys	-	Retorna una vista las claves en el diccionario
values	-	Retorna una vista los valores en el diccionario
items	-	Retorna una vista las clave-valor en el diccionario
get	key	Retorna el valor asociado a la clave o None
get	key, alt	Retorna el valor asociado a la clave o el valor <i>alt</i>

Los métodos de los diccionarios usan notación de puntos, que especifica el nombre del método a la derecha del punto y el nombre del objeto en el que aplicar el método inmediatamente a la izquierda del punto. Los paréntesis vacíos en el caso de las claves indican que este método no toma parámetros.

Nivel II: Representación, proceso y visualización de Datos

```
list(inventario.keys())  
['manzanas', 'naranjas', 'peras']  
  
for key in inventario.keys():  
    print("Para la clave", key, "el valor es: ",  
          inventario[key])  
  
Para la clave manzanas el valor es: 230  
Para la clave naranjas el valor es: 300  
Para la clave peras el valor es: 0  
  
list(inventario.keys())[0] # muestra el primer elemento  
'manzanas'
```

También podemos utilizar

```
list(inventario)  
['manzanas', 'naranjas', 'peras']  
  
for key in inventario:  
    print(key)  
  
manzanas  
naranjas  
peras
```

El método de `keys` devuelve las claves, no necesariamente en el mismo orden en que se agregaron al diccionario o en cualquier otro orden en particular.

Los métodos de `values` y `items` son similares a `keys`. Devuelven los objetos que se pueden iterar.

```
list(inventario.items())  
[('manzanas', 230), ('naranjas', 300), ('peras', 0)]  
  
list(inventario.values())  
[230, 300, 0]
```

Los elementos que retorna `items` son tuplas que contienen la clave y el valor asociado.

```
for key, value in inventario.items():  
    print(key, value)  
  
manzanas 230  
peras 150  
naranjas 300
```

Los operadores `in` y `not in` pueden comprobar si una clave está en el diccionario:

Nivel II: Representación, proceso y visualización de Datos

```
print('manzanas' in inventario)
print('cerezas' in inventario)
True
False
```

Este operador puede resultar muy útil, porque al buscar una clave inexistente en un diccionario provoca un error de tiempo de ejecución.

```
if 'manzanas' in inventario:
    print(' Hay {0} manzanas en la
          fruteria'.format(inventario['manzanas']))
else:
    print('No hay manzanas')
230
```

El método `get` nos permite acceder al valor asociado con una clave, similar al operador `[]`. La diferencia importante es que `get` no provocará un error en tiempo de ejecución si la clave no está presente. En su lugar, devolverá `None`. Existe una variación de `get` que permite un segundo parámetro que sirve como valor de retorno alternativo en el caso de que la clave no esté presente. Veamos un ejemplo:

```
inventario = {"manzanas":230, "peras":150, "naranjas":300}
print(inventario.get("manzanas"))
230

print(inventario.get("cerezas"))
None

print(inventario.get("cerezas",0))
0
```

Alias y copia

Debido a que los diccionarios son mutables, debe tener en cuenta el uso de alias (también ocurre con las listas). Siempre que dos variables se refieren al mismo objeto de diccionario, los cambios en una afectan a la otra.

```
opuestos = {'arriba': 'abajo', 'bueno': 'malo',
            'verdadero': 'falso'}
alias = opuestos
alias
{'arriba': 'abajo', 'bueno': 'malo', 'verdadero': 'falso'}

print(alias is opuestos)
True

# Insertamos una nueva clave:valor en alias
alias['derecha'] = 'izquierda'
print(opuestos['derecha'])
```

Nivel II: Representación, proceso y visualización de Datos

```
print(alias['derecha'])
izquierda
izquierda
```

Observamos que al insertar una nueva `clave:valor` en `alias`, también afectó a `opuestos`, porque hacen referencia al mismo objeto.

```
print(opuestos)
print(alias)
{'arriba': 'abajo', 'bueno': 'malo', 'verdadero': 'falso',
 'derecha': 'izquierda'}
{'arriba': 'abajo', 'bueno': 'malo', 'verdadero': 'falso',
 'derecha': 'izquierda'}
```

Si deseamos modificar un diccionario y conservar una copia del original, debemos utilizar el método `copy`. De esta forma, a la copia del diccionario no le afectarán los cambios que se realicen al original. Veamos un ejemplo:

```
copia = opuestos.copy()
print(copia)
{'arriba': 'abajo', 'bueno': 'malo', 'verdadero': 'falso',
 'derecha': 'izquierda'}

copia['blanco'] = 'negro' # no cambia opuestos
print("Copia: ", copia)
print("Opuestos: ", opuestos)

Copia: {'arriba': 'abajo', 'bueno': 'malo', 'verdadero':
 'falso', 'derecha': 'izquierda', 'blanco': 'negro'}
Opuestos: {'arriba': 'abajo', 'bueno': 'malo', 'verdadero':
 'falso', 'derecha': 'izquierda'}
```

Acumulación de varios resultados en un diccionario

Hasta ahora para generar un acumulador; debíamos recorrer los elementos en una secuencia, actualizando una variable para cada acumulador.

En lugar de acumular un solo resultado, utilizando diccionarios es posible acumular muchos resultados. Ahora validaremos cuáles son las letras que se utilizan con más frecuencia en inglés. Utilizaremos el texto de la historia de Sherlock Holmes, "A Study in Scarlet", de Sir Arthur Conan Doyle.

```
f = open('scarlet.txt', 'r')
texto = f.read()

t_contador = 0
s_contador = 0

for c in texto: # lee caracter por caracter
```

```
if c == 't':
    t_contador = t_contador + 1
elif c == 's':
    s_contador = s_contador + 1

print("Hay " + str(t_contador) + " ocurrencias de t")
print("Hay " + str(s_contador) + " ocurrencias de s")
```

en este primer ejemplo, utilizamos un acumulador para contar la ocurrencia de t y otro para la de s, tendríamos que hacer esto mismo para cada letra del abecedario. Un diccionario puede contener todas las variables del acumulador. Cada clave en el diccionario será una letra, y el valor será el número de veces que aparece la letra en el texto.

```
f = open('scarlet.txt', 'r')
texto = f.read()

contador_letras = {} # inicializamos el diccionario

for letra in texto.lower():
    if letra not in contador_letras:
        # si la letra no se encuentra, creamos la clave con valor 0
        contador_letras[letra] = 0

    # Incrementamos el valor, para la clave
    contador_letras[letra] = contador_letras[letra] + 1

# Mostramos los valores del diccionario
for c in contador_letras.keys():
    print(c + ": " + str(contador_letras[c]) + " ocurrencias")
```

```
t: 18283 ocurrencias
h: 13438 ocurrencias
e: 25680 ocurrencias
: 43507 ocurrencias
p: 3540 ocurrencias
r: 12235 ocurrencias
o: 15699 ocurrencias
j: 330 ocurrencias
c: 5306 ocurrencias
g: 4252 ocurrencias
...
```

Así como hemos iterado a través de los elementos de una lista para acumular un resultado, también podemos iterar a través de las claves en un diccionario, acumulando un resultado que puede depender de los valores asociados con cada una de las claves.

Por ejemplo, suponga que quisiéramos calcular una puntuación de Scrabble para el texto Study in Scarlet. Cada aparición de la letra "e" gana 1 punto, pero "q" gana 10 puntos. Utilizaremos un segundo diccionario, almacenado en la variable `valores_letras`.

Nivel II: Representación, proceso y visualización de Datos

```
valores_letras = {'a':1, 'b':3, 'c':3, 'd':2, 'e':1, 'f':4,
                  'g':2, 'h':4, 'i':1, 'j':8, 'k':5, 'l':1,
                  'm':3, 'n':1, 'o':1, 'p':3, 'q':10, 'r':1,
                  's':1, 't':1, 'u':1, 'v':4, 'w':4, 'x':8,
                  'y':4, 'z':10}

f = open('scarlet.txt', 'r')
texto = f.read()

contador_letra = {}

for letra in texto.lower():
    if letra not in contador_letra:
        contador_letra[letra] = 0

    contador_letra[letra] = contador_letra[letra] + 1

puntuacion = 0

for l in contador_letra:
    if l in valores_letras:
        #print(valores_letras[l])

        puntuacion = puntuacion + valores_letras[l] *
            contador_letra[l]

print("Puntuación total:", puntuacion)

Puntuación total: 350670
```

Cuando usar un diccionario

Las siguientes pautas nos ayudarán a reconocer cuándo un diccionario será beneficioso:

- Cuando un dato consta de un conjunto de propiedades de un solo elemento, un diccionario suele ser mejor. Por ejemplo: El código postal puede estar en el índice 2 en una lista, pero será más fácil de leer: `mydiction['zipcode']` que `mylst[2]`.
- Cuando tenemos una colección de pares de datos y, a menudo, tendrá que buscar uno de los pares según su primer valor, es mejor usar un diccionario que una lista de tuplas (clave, valor).
- Con un diccionario, podemos encontrar el valor de cualquier tupla (clave, valor) buscando la clave. Con una lista de tuplas, necesitaríamos recorrer la lista, examinando cada par para ver si tiene la clave que deseamos.
- Por otro lado, si tenemos una colección de pares de datos donde varios pares comparten el mismo primer elemento de datos, entonces no puede usar un diccionario, porque un diccionario requiere que todas las claves sean distintas entre sí.

Ordenar Diccionarios

Las claves del diccionario no están ordenadas en ningún orden en particular. De hecho, puede obtener un orden de salida diferente al de otra persona que ejecuta el mismo código. Podemos forzar que los resultados se muestren en un orden fijo, ordenando las claves.

Veamos antes la sintaxis de la función `sorted()`, que podemos utilizar en diccionarios o en cualquier objeto iterable.

Función Sorted ()

La función `sorted()`, retorna una nueva lista ordenada a partir de los elementos en iterable. Su sintaxis:

```
sorted(iterable, *, key=None, reverse=False)
```

Tiene dos argumentos opcionales que deben ser especificados como argumentos de palabra clave.

key: especifica una función de un argumento que es empleada para extraer una clave de comparación de cada elemento en iterable (por ejemplo, `key=str.lower`). El valor por defecto es `None` (compara los elementos directamente).

Reverse: es un valor boleado. Si es `True`, entonces la lista de elementos se ordena como si cada comparación fuera reversa.

La función `sorted()` es de gran ayuda para ordenar en múltiples pases (por ejemplo, ordenar por departamento, después por salario).

Observamos que tanto `list.sort()` como `sorted()` tienen un parámetro clave para especificar una función (u otra invocable) que se llamará en cada elemento de la lista antes de hacer comparaciones.

Veamos ahora, algunas diferencias entre `sort()` y `sorted()`

- El método `sort()` solo aplica para las listas. En contraste, la función `sorted()` acepta cualquier iterable.
- El método `sort()` modifica la lista in situ (y retorna `None` para evitar confusiones). Mientras que `sorted()` crea una nueva lista ordenada.

Retomemos ahora el ejemplo donde calculamos la ocurrencia de las letras en el texto *Study in Scarlet*, pero ahora ordenemos el diccionario por su clave:

```
f = open('scarlet.txt', 'r')
texto = f.read()

contador_letras = {} # inicializamos el diccionario
```

```
for letra in texto.lower():
    if letra not in contador_letras:
        # si la letra no se encuentra, creamos la clave con valor 0
        contador_letras[letra] = 0

    # Incrementamos el valor, para la clave
    contador_letras[letra] = contador_letras[letra] + 1

# Ordenamos el contador de letras y genera la lista ordenado

ordenado = sorted(contador_letras.keys())
print(ordenado)

# Mostramos los valores del diccionario
for c in ordenado:
    print(c + ": " + str(contador_letras[c]) + " ocurrencias")
```

```
...
a: 16261 ocurrencias
b: 2897 ocurrencias
c: 5306 ocurrencias
d: 9218 ocurrencias
e: 25680 ocurrencias
f: 4429 ocurrencias
g: 4252 ocurrencias
h: 13438 ocurrencias
i: 14053 ocurrencias
j: 330 ocurrencias
k: 1500 ocurrencias
...
```

 **CLOUD**
FORMACIÓN

Ahora ordenaremos el diccionario por el número de ocurrencias, para ello utilizaremos una función lambda como parámetro key de la función sorted() y el parámetro reverse = True.

La función lambda: lambda k: contador_letras[k] retorna la cantidad de ocurrencias de una letra, al pasarla como parámetro de key, ordenará la lista por ese valor.

```
f = open('scarlet.txt', 'r')
texto = f.read()

contador_letras = {} # inicializamos el diccionario

for letra in texto.lower():
    if letra not in contador_letras:
        # si la letra no se encuentra, creamos la clave con valor 0
        contador_letras[letra] = 0

    # Incrementamos el valor, para la clave
    contador_letras[letra] = contador_letras[letra] + 1
```



```
# Ordenamos el contador de letras y genera la lista ordenado
ordenado = sorted(contador_letras.keys(),
                  key = lambda k: contador_letras[k],
                  reverse=True)

# Mostramos los valores del diccionario
for c in ordenado:
    print(c + ": " + str(contador_letras[c]) + " ocurrencias")
```

```
      : 43507 ocurrencias
e: 25680 ocurrencias
t: 18283 ocurrencias
a: 16261 ocurrencias
o: 15699 ocurrencias
i: 14053 ocurrencias
n: 13954 ocurrencias
h: 13438 ocurrencias
r: 12235 ocurrencias
s: 12234 ocurrencias
d: 9218 ocurrencias
l: 7701 ocurrencias
u: 5747 ocurrencias
c: 5306 ocurrencias
m: 5288 ocurrencias
```

Otra forma es invocando a una función, que retorne el valor para una clave dada.

```
f = open('scarlet.txt', 'r')
texto = f.read()

contador_letras = {} # inicializamos el diccionario

def obtener_valor(clave):
    return contador_letras[clave]

for letra in texto.lower():
    if letra not in contador_letras:
        # si la letra no se encuentra, creamos la clave con valor 0
        contador_letras[letra] = 0

    # Incrementamos el valor, para la clave
    contador_letras[letra] = contador_letras[letra] + 1

# Ordenamos el contador de letras y genera la lista ordenado
ordenado = sorted(contador_letras.keys(),
                  key=obtener_valor,
                  reverse=True)
```

```
# Mostramos los valores del diccionario
for c in ordenado:
    print(c + ": " + str(contador_letras[c]) + " ocurrencias")
```

Ordenar colecciones por más de una propiedad

¿Qué sucede cuando dos elementos están "vinculados" en el orden de clasificación? Por ejemplo, supongamos que ordenamos una lista por la longitud de su palabra. ¿Qué palabra de cinco letras aparecerá primero?

La respuesta es que el intérprete de Python clasificará los elementos vinculados en el mismo orden en que estaban antes de la clasificación.

¿Qué pasaría si quisiéramos ordenarlos alfabéticamente, cuando las palabras tuvieran la misma longitud? Python nos permite especificar múltiples condiciones cuando realizamos una ordenación:

Hay un orden de clasificación integrado, si no especificamos ninguna función clave. Para los números, es de menor a mayor. Para cadenas, es orden alfabético. Para una secuencia de tuplas, el orden de clasificación predeterminado se basa en el orden de los primeros elementos de la tupla, luego los segundos elementos, luego los terceros elementos y así sucesivamente.

Ordenar una tupla

```
tuplas = [('A', 3, 2),
          ('C', 1, 4),
          ('B', 3, 1),
          ('A', 2, 4),
          ('C', 1, 2)]
```

```
for tupla in sorted(tuplas):
    print(tupla)
```

```
('A', 2, 4)
('A', 3, 2)
('B', 3, 1)
('C', 1, 2)
('C', 1, 4)
tuplas_ordenadas = sorted(tuplas)
print("Original: ", tuplas)
print("Nueva:     ", tuplas_ordenadas)
```

```
Original: [('A', 3, 2), ('C', 1, 4), ('B', 3, 1), ('A', 2, 4), ('C', 1, 2)]
Nueva:    [('A', 2, 4), ('A', 3, 2), ('B', 3, 1), ('C', 1, 2), ('C', 1, 4)]
```

Ordenar una lista

Ahora, vamos a ordenar una lista de palabras primero por su longitud, de menor a mayor, y luego alfabéticamente para romper los lazos entre palabras de la misma longitud. Para hacer eso, en el parámetro `key` pasamos una función lambda que devuelve una tupla cuyo primer elemento es la longitud de la palabra y el segundo elemento es el nombre en sí.

```
frutas = ['pera', 'manzana', 'arándanos',  
          'mango', 'kiwi', 'cerezas']  
  
# ordena por la longitud y luego por el nombre  
  
frutas_ordenadas = sorted(frutas,  
                           key=lambda  
                               nombre:(len(nombre), nombre))  
  
print("Original: ", frutas)  
print("Ordenada: ", frutas_ordenadas)  
Original: ['pera', 'manzana', 'arándanos', 'mango', 'kiwi',  
          'cerezas']  
Ordenada: ['kiwi', 'pera', 'mango', 'cerezas', 'manzana',  
          'arándanos']
```

Aquí, cada palabra se evalúa primero por su longitud y luego por su orden alfabético. Podríamos continuar especificando otras condiciones al incluir más elementos en la tupla.

Ahora vamos a ordenarlo de mayor a menor y luego por orden alfabético. Sólo necesitamos agregar el parámetro `reverse=True` en la función `sorted()`:

```
frutas_ordenadas = sorted(frutas, key=lambda nombre:(len(nombre),  
nombre), reverse=True)
```

```
Original: ['pera', 'manzana', 'arándanos', 'mango', 'kiwi', 'cerezas']  
Ordenada: ['arándanos', 'manzana', 'cerezas', 'mango', 'pera', 'kiwi']
```

No solo ordena las palabras de mayor a menor, sino también en orden alfabético inverso.

Ordenar un diccionario

Comenzaremos ordenando el diccionario estados por orden alfabético:

```
estados = {"Minnesota": ["St. Paul", "Minneapolis",  
                          "Saint Cloud", "Stillwater"],  
           "Michigan": ["Ann Arbor", "Traverse City", "Lansing",  
                        "Kalamazoo"],  
           "Washington": ["Seattle", "Tacoma", "Olympia",  
                           "Vancouver"]}  
  
ordenado = sorted(estados) # Ordena alfabéticamente
```

Nivel II: Representación, proceso y visualización de Datos

```
for estado in ordenado:
    print(estado, ": ", estados[estado])
```

```
Michigan: ['Ann Arbor', 'Traverse City', 'Lansing', 'Kalamazoo']
Minnesota: ['St. Paul', 'Minneapolis', 'Saint Cloud', 'Stillwater']
Washington: ['Seattle', 'Tacoma', 'Olympia', 'Vancouver']
```

Ahora ordenaremos por la primera ciudad de cada estado:

Ordena por la longitud de la primera ciudad del estado

```
estados_ordenados = sorted(estados,
                            key=lambda estado:
                            len(estados[estado][0]))
```

```
for estado in estados_ordenados:
    print(estado, ": ",
          estados[estado][0],
          len(estados[estado][0]))
```

```
Washington: Seattle 7
Minnesota: St. Paul 8
Michigan: Ann Arbor 9
```

Ordenaremos utilizando una función que cuenta el número de ciudades que comienzan con S:

Función que cuenta las ciudades que comienzan por S

```
def contar_ciudades_s(l_ciudades):
    ct = 0
    for ciudad in l_ciudades:
        if ciudad[0] == "S":
            ct += 1
    return ct
```

```
estados_ordenados = sorted(estados, key=lambda estado:
                           contar_ciudades_s(estados[estado]))
```

```
for estado in estados_ordenados:
    print(estado, ": ", estados[estado], "No. ciudades:",
          contar_ciudades_s(estados[estado]))
```

```
Michigan:['Ann Arbor', 'Traverse City', 'Lansing', 'Kalamazoo'], No. ciudades: 0
Washington:['Seattle', 'Tacoma', 'Olympia', 'Vancouver'], No. ciudades: 1
Minnesota:['St. Paul', 'Minneapolis', 'Saint Cloud', 'Stillwater'], No. ciudades: 3
```

Veamos un ejemplo, utilizando diccionarios anidados:

```
clima = {'Reykjavik':{'temp':15, 'condicion': 'lluvias'},
        'Buenos Aires': {'temp': 13, 'condicion': 'nublado'},
```

```
'Cairo': {'temp': 36, 'condicion': 'soleado'},  
'Berlin': {'temp': 32, 'condicion': 'soleado'},  
'Caloocan': {'temp': 26, 'condicion': 'soleado'}}
```

```
clima
```

```
{'Reykjavik': {'temp': 15, 'condicion': 'lluvias'},  
 'Buenos Aires': {'temp': 13, 'condicion': 'nublado'},  
 'Cairo': {'temp': 36, 'condicion': 'soleado'},  
 'Berlin': {'temp': 32, 'condicion': 'soleado'},  
 'Caloocan': {'temp': 26, 'condicion': 'soleado'}}
```

```
# Ordenar alfabéticamente
```

```
clima_ordenado = sorted(clima.items())  
clima_ordenado
```

```
[('Berlin', {'temp': 32, 'condicion': 'soleado'}),  
 ('Buenos Aires', {'temp': 13, 'condicion': 'nublado'}),  
 ('Cairo', {'temp': 32, 'condicion': 'soleado'}),  
 ('Caloocan', {'temp': 26, 'condicion': 'soleado'}),  
 ('Reykjavik', {'temp': 15, 'condicion': 'lluvias'})]
```

```
for s in clima_ordenado:  
    print(s[0], clima[s[0]]['temp'], "°C",  
          clima[s[0]]['condicion'])
```

```
Berlin 32°C soleado  
Buenos Aires 13°C nublado  
Cairo 32°C soleado  
Caloocan 26°C soleado  
Reykjavik 15°C lluvias
```

```
# Ordenar por temperatura
```

```
clima_ordenado = sorted(clima,  
                        key=lambda c: (clima[c]['temp']),  
                        reverse=True)  
print(clima_ordenado)  
['Cairo', 'Berlin', 'Caloocan', 'Reykjavik', 'Buenos Aires']
```

```
for s in clima_ordenado:  
    print(f"La temperatura en la ciudad {s},  
          es de {clima[s]['temp']} °C -  
          {clima[s]['condicion']}")
```

```
La temperatura en la ciudad Cairo, es de 32 °C - soleado  
La temperatura en la ciudad Berlin, es de 32 °C - soleado
```

La temperatura en la ciudad Caloocan, es de 26 °C - soleado
La temperatura en la ciudad Reykjavik, es de 15 °C - lluvias
La temperatura en la ciudad Buenos Aires, es de 13 °C - nublado

Ordenar por temperatura y ciudad

```
clima_ordenado = sorted(clima, key=lambda c:  
                        (clima[c]['temp'],c))  
print(clima_ordenado)
```

```
for s in clima_ordenado:  
    print(f"La temperatura en la ciudad {s},  
          es de {clima[s]['temp']} °C -  
          {clima[s]['condicion']}")
```

La temperatura en la ciudad Buenos Aires, es de 13 °C - nublado
La temperatura en la ciudad Reykjavik, es de 15 °C - lluvias
La temperatura en la ciudad Caloocan, es de 26 °C - soleado
La temperatura en la ciudad Berlin, es de 32 °C - soleado
La temperatura en la ciudad Cairo, es de 32 °C - soleado

Generadores

Son funciones que nos permitirán obtener sus resultados a medida que los necesitamos. Es decir, cada vez que llamemos a la función generadora, nos retornará un nuevo resultado.

Estas funciones generan datos en tiempo de ejecución. Además, también podemos acelerar búsquedas y crear bucles más rápidos.

Los generadores son en realidad iteradores, pero sólo permiten ser iterados una vez. Esto se debe a que no almacenan todos los valores en memoria, sino que los van generando en tiempo real. Pueden ser usados de dos formas diferentes, iterándolos con un bucle `for` o pasándolos a una función.

Para construir generadores sólo tenemos que usar la orden `yield`. Esta orden devolverá un valor (igual que hace `return`), además, congelará la ejecución de la función hasta la próxima vez que le pidamos un valor.

```
def funcion_generadora():  
    for i in range(10):  
        yield i  
  
for item in funcion_generadora():  
    print(item)
```

0
1
2

3
4
5
6
7
8
9

Esta función generadora genera un número de 0 a 9, cada vez que es llamada. Los generadores son más útiles cuando necesitamos realizar cálculos para un número muy elevado de elementos. Es decir, cuando no queremos tener en memoria todos los elementos a la vez.

En el siguiente ejemplo, tenemos una función que genera un número Fibonacci cada vez que es invocada.

```
def fibon(n):  
    a = b = 1  
    for i in range(n):  
        yield a  
        a, b = b, a + b
```

```
for x in fibon(10):  
    print(x)
```

1
1
2
3
5
8
13
21
34
55

CLOUD
FORMACIÓN

Veamos otros ejemplos:

```
def pares():  
    index = 1  
  
    while True:  
        yield index*2  
        index = index + 1
```

Para probarlo simplemente iteramos sobre la función, en este caso sólo invocaremos 10 veces a la función generadores de números pares.

```
c = 0  
for i in pares():  
    print(i)
```

Nivel II: Representación, proceso y visualización de Datos

```
c += 1
if c == 10:
    break
2
4
6
8
10
12
14
16
18
20
```

La siguiente función generadora, crea los múltiplos de una numero dado:

```
def multiplos_de(n):
    index = 1
    while True:
        yield index*n
        index = index + 1
```

Para probarla simplemente iteramos sobre la función, en este caso sólo invocaremos 6 veces a la función generadores de los múltiplos de 10.

```
c = 0
n = int(input("Introduzca el número para generar sus múltiplos:
"))
for i in multiplos_de(n):
    print(i)
    c += 1
    if c == 6:
        break
```

```
Introduzca el número para generar sus múltiplos: 10
10
20
30
40
50
60
```

Como vimos en unidades anteriores la función `next()`, nos permite acceder al siguiente elemento de una secuencia.

```
def funcion_generadora():
    for i in range(3):
        yield i
```



```
gen = funcion_generadora()
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
```

```
0
1
2
```

```
-----
StopIteration                                Traceback (most recent call last)
<ipython-input-402-e3a07ab95b5b> in <module>
      7 print(next(gen))
      8 print(next(gen))
----> 9 print(next(gen))
```

StopIteration:

Como podemos ver, cuando se llega al final de la función, si se intenta llamar otra vez al next() tendremos un error `StopIteration`, ya que no hay más valores. Esto se debe a que la función no tiene más valores de los que hacer `yield`, es decir se ha llegado al final.

En el siguiente ejemplo, simularemos un sistema de tickets, cada vez que llega un cliente este tomará un ticket:

```
def generar_ticket():
    ticket = 1
    while True:
        yield ticket
        ticket += 1

gen = generar_ticket()
t = ""
```

```
while t == "" :
    t = input("Tome un número: ").upper()
    if t == "":
        print(next(gen))
```

```
Tome un número:
1
Tome un número:
2
Tome un número:
3
```

En el siguiente ejemplo, se crea un generador al crear una lista por comprensión, que obtiene las “stop words” de una lista de palabras:

```
f = open('stopwords_español.txt', 'r', encoding="utf8")
texto = f.read()
```

Nivel II: Representación, proceso y visualización de Datos

```
palabras = texto.split()

gen = (p for p in palabras)

print(next(gen))
print(next(gen))
print(next(gen))
a
al
algo
```

Podemos recorrer todo el generador utilizando un for.

```
for p in gen:
    print(p)
```

```
algunas
algunos
ante
antes
como
con
contra
cual
cuando
```

Una finalizado el recorrido del generador, si utilizamos `next()`, este retornará un error y ejecutamos un bucle para recorrer el generador, no retornará ningún valor. Esto sucede porque **los objetos generadores no son reutilizables**. Debemos obtener un nuevo generador ejecutando una función generadora o ejecutando una nueva expresión por comprensión.

Consultas

Una de las principales características de Python es que este es un lenguaje multiparadigma. Podemos desarrollar proyectos utilizando un enfoque orientado a objetos, imperativo e inclusive funcional.

Existen diferentes módulos que hacen uso de programación funcional para aplicar operaciones de consultas y transformaciones sobre conjunto de datos, especialmente importante para el procesamiento de Big Data:

- **Intertools:** módulo integrado con Python, que ofrece un gran número de herramientas para facilitarnos la creación de iteradores.
- **Operator:** módulo integrado con Python, donde podemos encontrar los principales operadores de Python convertidos en funciones.
- **Functools:** integrado con Python, y ayuda crear fabricantes de funciones puras.

Nivel II: Representación, proceso y visualización de Datos

- **Fn:** este módulo, creado por Alexey Kachayev, ofrece herramientas para aplicar programación funcional de una forma sencilla.
- **Cytoolz:** modulo creado por Erik Welch que nos proporciona varias herramientas para la Programación Funcional, especialmente orientado a operaciones de análisis de datos funcionales de alto rendimiento con tipos integrados como list y dict, y estructuras de datos definidas por el usuario.
- **Macropy:** modulo creado por Li Haoyi, que trae a Python características propias de los lenguajes puramente funcionales, como son: pattern matching, tail call optimization, y case classes.

Funciones

A continuación, vamos a repasar algunas de las funciones más populares en cuanto a programación funcional.

`map()` : aplica una función a cada uno de los elementos de una lista.

```
enteros = [1, 2, 4, 7]
cuadrados = list(map(lambda x : x ** 2, enteros))
print(cuadrados)
[1, 4, 16, 49]
```

`filter()` : filtra una lista de elementos para los que una función devuelve True.

```
valores = [1, 2, 3, 4, 5, 6, 7, 8, 9]
impares = list(filter(lambda x : x % 2 != 0, valores))
print(impares)
[1, 3, 5, 7, 9]
```

`reduce()` : Esta función se utiliza principalmente para llevar a cabo un cálculo acumulativo sobre una lista de valores y devolver el resultado. La función `reduce()` está incluida en el módulo `functools`.

```
from functools import reduce

valores = [2, 4, 6, 5, 4]

suma = reduce(lambda x, y: x + y, valores)
print(suma)
21
```

Otras funciones para objetos iterables

`zip()`

La función `zip()` de Python se define como `zip(*iterables)`. La función toma un iterable como argumento y devuelve un iterador. Este iterador genera una serie de tuplas que contienen elementos de cada iterable. `zip()` puede aceptar cualquier tipo de iterable, como archivos, listas, tuplas, diccionarios, conjuntos, etc.

Nivel II: Representación, proceso y visualización de Datos

Si utilizamos `zip()` con `n` argumentos, la función devolverá un iterador que genera tuplas de longitud `n`. Para ver esto en acción, eche un vistazo al siguiente bloque de código:

```
numeros = [1, 2, 3]
letras = ['a', 'b', 'c']

zipped = zip(numeros, letras)

zipped
<itertools.zip_longest at 0x1f26e77cc20>

type(zipped)
itertools.zip_longest

list(zipped)
[(1, 'a'), (2, 'b'), (3, 'c')]
```

En el ejemplo, utilizamos `zip(números, letras)` para crear un iterador que produce tuplas de la forma `(x, y)`. En este caso, los valores `x` se toman de los números y los valores `y` se toman de las letras. Observe cómo la función `zip()` de Python devuelve un iterador. Para recuperar el objeto de lista final, debe utilizar `list()` para consumir el iterador.

```
x = [1, 4, 7, 8, 9]
y = [-3, 6, 10, 3, -2]
z = [2, 3, 5, 6, -1]

datos = zip(x, y, z)
list(datos)

[(1, -3, 2), (4, 6, 3), (7, 10, 5), (8, 3, 6), (9, -2, -1)]
```

En el ejemplo, utilizamos `zip(x, y, z)` para crear un iterador que produce tuplas de la forma `(x, y, z)`.

Desempaquetar

Si necesitamos separar los elementos de cada tupla en secuencias independientes, podemos utilizar `zip()` junto con el operador de desempaquete `*`, veamos un ejemplo, donde queremos obtener separadamente los valores de `x` e `y` de la variable `datos`:

```
datos = [(43, 34), (76, 45), (34, 34), (63, 23), (56, 43),
         (82, 76), (87, 26), (55, 18), (64, 24), (87, 74),
         (95, 23), (23, 56), (14, 23), (65, 23), (67, 34),
         (25, 56), (23, 32), (85, 23)]

x, y = zip(*list(datos))
```

```
x
```

Nivel II: Representación, proceso y visualización de Datos

(43, 76, 34, 63, 56, 82, 87, 55, 64, 87, 95, 23, 14, 65, 67,
25, 23, 85)

y
(34, 45, 34, 23, 43, 76, 26, 18, 24, 74, 23, 56, 23, 23, 34,
56, 32, 23)

La función `zip()` de Python nos permite iterar en paralelo a través de dos o más iterables. Puesto que `zip()` genera tuplas, podemos desempaquetar éstos en el encabezado de un bucle `for`:

```
numeros = [1, 2, 3]
letras = ['a', 'b', 'c']

for l, n in zip(letras, numeros):
    print(f'letras: {l}')
    print(f'numeros: {n}')
```

```
letras: a
numeros: 1
letras: b
numeros: 2
letras: c
numeros: 3
```

Veamos un ejemplo, donde necesitamos graficar las ubicaciones de pisos en alquiler.

```
import matplotlib.pyplot as plt

# Data
ubicaciones = [(43.374005, -8.43042), (43.385741, -8.406578),
               (43.382966, -8.396042), (43.378443, -8.402415)]

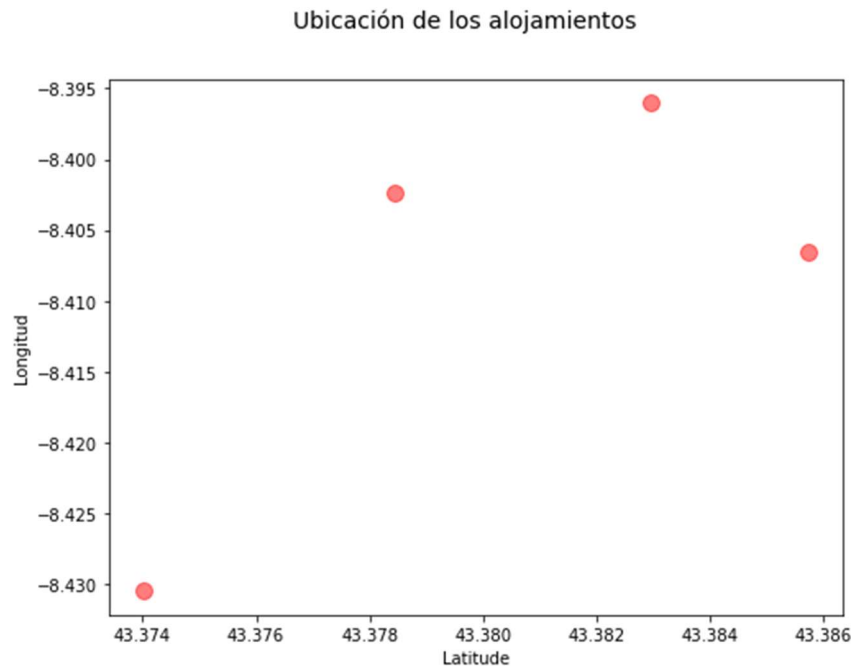
x, y = zip(*ubicaciones)
# Generamos el gráfico

fig, ax = plt.subplots(1, figsize=(10, 6))
fig.suptitle('Ubicación de los alojamientos')

ax.scatter(x, y,
           color="red",
           s=100,
           alpha=0.5,
           linewidths=1)

ax.set_xlabel('Latitude')
ax.set_ylabel('Longitud')
```

```
# Mostrar el gráfico
plt.show()
```



Pasar argumentos de longitud desigual

Cuando trabajemos con la función `zip()` de Python, es importante prestar atención a la longitud de sus iterables. Es posible que las iterables que se pasan como argumentos no sean de la misma longitud.

En estos casos, el número de elementos que `zip()` retorna será igual a la longitud del iterable más corto. Los elementos restantes en los iterables más largos serán totalmente ignorados por `zip()`:

```
list(zip(range(5), range(100)))
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4)]
```

Si los valores finales o no coincidentes son importantes, podemos utilizar `itertools.zip_longest()` en lugar de `zip()`. Con esta función, los valores que faltan se reemplazarán con el valor que pasemos al argumento `fillvalue` (el valor predeterminado es `None`). La iteración continuará hasta que se agote el iterable más largo:

```
from itertools import zip_longest

numbers = [1, 2, 3, 4]
letters = ['a', 'b', 'c']
longest = range(5)

zipped = zip_longest(numbers, letters, longest, fillvalue='?')
list(zipped)
```

```
[(1, 'a', 0), (2, 'b', 1), (3, 'c', 2), (4, '?', 3),  
 ('?', '?', 4)]
```

`all(iterable)`: Devuelve True si todos los elementos de la iterable son true (o si el iterable está vacío).

```
numbers = [1, 2, 3, 4]  
all(numbers)  
True
```

```
numbers = [1, 2, 3, 4, ""]  
all(numbers)  
False
```

```
numbers = [1, 2, 3, 4, None]  
all(numbers)  
False
```

```
numbers = []  
all(numbers)  
True
```

`any(iterable)`: Devuelve True si cualquier elemento de la iterable es true. Si el iterable está vacío, devuelve False.

```
numbers = [1, 2, 3, 4]  
any(numbers)  
True
```

```
numbers = [1, 2, 3, 4, ""]  
any(numbers)  
True
```

```
numbers = [1, 2, 3, 4, None]  
any(numbers)  
True
```

```
numbers = []  
any(numbers)  
False
```

NOTA: Python considera que todo lo que no sea False, None o un string vacío es True.

`max(iterable, *[, key, default])`: Devuelve el elemento más grande en un iterable o el más grande de dos o más argumentos.

```
max("abcDEF") # cadena o string  
c
```

```
max([2, 1, 4, 3]) # lista
4
```

```
max(("one", "two", "three")) # tuple
two
```

```
max({1: "one", 2: "two", 3: "three"}) # diccionario
3
```

```
max([])
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-135-a48d8f8c12de> in <module>
----> 1 max([])

ValueError: max() arg is an empty sequence
```

```
max([], default=0)
0
```

```
max("c", "b", "a", "Y", "Z", key=str.lower)
Z
```

`min(iterable, *[, key, default])`: Devuelve el elemento más pequeño en un iterable o el más pequeño de dos o más argumentos.

```
min("abcDEF") # cadena o string
D
```

```
min([2, 1, 4, 3]) # lista
1
```

```
min(("one", "two", "three")) # tuple
one
```

```
min({1: "one", 2: "two", 3: "three"}) # diccionario
3
```

```
min([])
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-146-cb4199d4da81> in <module>
----> 1 min([])

ValueError: min() arg is an empty sequence
```

```
min([], default=0)
0
```


Nivel II: Representación, proceso y visualización de Datos

```
min("c", "b", "a", "Y", "Z", key=str.lower)
a
```

`sum(iterable, start=0)`: Las sumas se inician y los elementos de un iterable de izquierda a derecha y devuelve el total. Los elementos de iterable son normalmente números y el valor inicial no se permite que sea una cadena.

```
numbers = [1, 2, 3, 4]
sum(numbers)
10

sum(numbers, start=5)
15

sum({1: "one", 2: "two", 3: "three"})
6
```

modulo Fn.py

Algunas de las cosas que nos ofrece este módulo son: Estructuras de datos inmutables, lambdas al estilo de Scala, nuevas Funciones de orden superior, entre otras.

Para instalar el módulo, desde el terminal ejecute:

```
pip install fn
```

Veamos algunos ejemplos:

```
from fn import _
```

Lambdas al estilo scala

```
(_ + _)(10, 3)
13
```

```
items = list(range(1,11))
```

```
cuadrados = map( _ ** 2, items)
list(cuadrados)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Funciones de orden superior

```
from fn import F
```

```
from operator import add, mul
```

Composición de funciones

```
F(add, 1)(10)
11
```

f es una función que llama a otra función.

<< operador de composición de funciones.

```
f = F(add, 5) << F(mul, 100)
```

```
""" cada valor de la lista primero se multiplica por 100 y
luego se le suma 5, según composición de f de arriba
"""
```

```
list(map(f, [0, 1, 2, 3]))
[5, 105, 205, 305]
```

Módulo cytoolz

Este módulo nos provee varias herramientas para trabajar con funciones, iteradores y diccionarios. Para instalar el módulo, desde el terminal ejecute:

```
pip install cytoolz
```

Funciones cytoolz

- `pipe()`: permite encadenar una serie de funciones sobre un conjunto de datos. Estas funciones son aplicadas en el orden en el cual se especifican en los argumentos.
- `filter()`: aplica un filtro sobre el conjunto de datos. Es similar a la función `filter()` integrada pero sin tener que especificar un iterable.
- `map()`: es similar a la función `map()` integrada pero sin tener que especificar un iterable.
- `get()`: permite especificar elementos en una secuencia o en un dict.
- `groupby()`: es una función que se aplica sobre cada elemento de la secuencia y que retorna la clave de agrupación, y el segundo argumento es la secuencia de elementos.
- `reduceby()`: Esta función aplica primero una agrupación y después una reducción sobre el diccionario resultante. Sus argumentos son

En el siguiente ejemplo, obtendremos el teléfono de los clientes cuyos nombres comience por "J"

```
clientes = [dict(id=1, nombre='Juan', telefono='111111111'),
            dict(id=2, nombre='Ana', telefono='222222222'),
            dict(id=3, nombre='Mario', telefono='333333333'),
            dict(id=4, nombre='Jose', telefono='444444444')]
```

```
from cytoolz.curried import pipe, map, filter, get
```

```
resultado = pipe(clientes,
```

```
filter(lambda c: c['nombre'][0]=='J'),  
map(get(['nombre', 'telefono'])) )  
  
print(list(resultado))  
[('Juan', '111111111'), ('Jose', '444444444')]
```

En el siguiente ejemplo agruparemos, por la primera letra del nombre:

```
from cytoolz import groupby  
  
grupos = groupby(lambda c: c['nombre'][0], clientes)  
print( grupos )  
  
{'J': [{ 'id': 1, 'nombre': 'Juan', 'telefono': '111111111'},  
       { 'id': 4, 'nombre': 'Jose', 'telefono': '444444444'}],  
 'A': [{ 'id': 2, 'nombre': 'Ana', 'telefono': '222222222'}],  
 'M': [{ 'id': 3, 'nombre': 'Mario', 'telefono': '333333333'}]}
```

Ordenando objetos con operator.

El módulo `operator`, tienen tres funciones que nos permiten ordenar todo tipo de objetos en forma muy sencilla, ellas son `itemgetter()`, `attrgetter()` y `methodcaller()`.

- `itemgetter(item, ...)`: retorna un objeto que recupera el valor de los elementos de los índices dados.
- `attrgetter(attr, ...)`: retorna un objeto que recuperar los valores de los atributos o claves dados.
- `methodcaller(name, ...)`: retorna un objeto que recupera lo que devuelve la ejecución de un método de nombre dado.

Veamos un ejemplo:

```
from operator import itemgetter, attrgetter, methodcaller  
  
class Estudiante:  
  
    def __init__(self, nombre, nota, edad):  
        self.nombre = nombre  
        self.nota = nota  
        self.edad = edad  
  
    def __repr__(self):  
        return repr((self.nombre, self.nota, self.edad))  
  
    def len_nombre(self):  
        return len(self.nombre)
```

Datos

```
estudiantesTuplas = [('Juan', 'A', 15), ('Julia', 'B', 12),  
('David', 'B', 10)]
```

```
estudiantesObj = [Estudiante('john', 'A', 15),  
Estudiante('jane', 'B', 12), Estudiante('dave', 'B', 10),]
```

```
estudiantesDic = [{'nombre': 'Juan', 'nota': 'A', 'edad': 15},  
{'nombre': 'Julia', 'nota': 'B', 'edad': 12},  
{'nombre': 'David', 'nota': 'B', 'edad': 10}]
```

Ordenar por edad

```
print( sorted(estudiantesTuplas, key=itemgetter(2)))
```

```
print( sorted(estudiantesObj, key=attrgetter('edad')))
```

```
print( sorted(estudiantesDic, key=itemgetter('edad')))
```

```
[('David', 'B', 10), ('Julia', 'B', 12), ('Juan', 'A', 15)]  
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]  
[{'nombre': 'David', 'nota': 'B', 'edad': 10}, {'nombre':  
'Julia', 'nota': 'B', 'edad': 12}, {'nombre': 'Juan', 'nota':  
'A', 'edad': 15}]
```

Ordenar por nota y edad

```
print( sorted(estudiantesTuplas, key=itemgetter(1, 2)))
```

```
print( sorted(estudiantesObj, key=attrgetter('nota', 'edad')))
```

```
print( sorted(estudiantesDic, key=itemgetter('nota', 'edad')))
```

```
[('Juan', 'A', 15), ('David', 'B', 10), ('Julia', 'B', 12)]  
[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]  
[{'nombre': 'Juan', 'nota': 'A', 'edad': 15}, {'nombre':  
'David', 'nota': 'B', 'edad': 10}, {'nombre': 'Julia', 'nota':  
'B', 'edad': 12}]
```

En el siguiente ejemplo, ordenaremos por la longitud del nombre del estudiante, invocando el método 'len_nombre' de la clase Estudiante:

```
print( sorted(estudiantesObj, key=methodcaller('len_nombre')))
```

```
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```