# 2 Estructuras de datos

Hasta aquí todo dato procesado, manipulado y operado ha sido almacenado en variables, sin embargo, para ciertos problemas no son suficientes. Supongamos un caso donde leemos una serie de temperaturas mensuales durante los últimos 10 años y que posteriormente queremos saber las temperaturas que han superado la media.

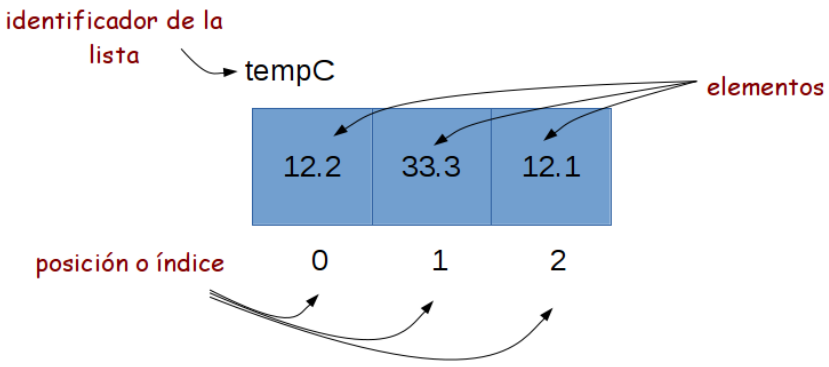
Si utilizamos variables, deberíamos leer los 120 valores para calcular el promedio y reingresar nuevamente las temperaturas mensuales para corroborar aquellas que superaron la media. Claramente el usuario de este programa no estará muy feliz de tener que reingresar la totalidad de los datos.

Para este tipo de problemas y muchos otros existen estructuras de datos más flexibles que las variables, que funcionan como contenedores de información.

En el presente capítulo haremos énfasis en dos de las estructuras comúnmente utilizadas como las *listas*, *diccionarios* y *tuplas* y, veremos con mayor detalle las *cadenas de caracteres*, ya presentadas en capítulos previos.

## 2.1 Listas

A diferencia de una variable que contiene un único dato por vez, una lista puede almacenar varios datos en forma simultánea en diferentes posiciones, por lo que para referirnos a uno de ellos necesitamos especificarle el índice o posición. Por ejemplo, en la siguiente lista denominada *tempC* hay almacenados tres valores numéricos flotantes, el primero está en la posición 0, el segundo en la posición 1 y, el tercero en la posición 2:




*Lista que contiene 3 valores*

Para inicializar la lista *tempC* con esos tres valores:

```
tempC = [12.2, 33.3, 12.1]   # Lista con 3 valores flotantes
vacia = []                   # Lista sin elementos
```

Para acceder a un elemento específico, debemos utilizar el identificador de la lista, seguido del índice entre corchetes (cualquier expresión entera), veamos un ejemplo donde realizamos las siguientes acciones:

1. Imprimir en pantalla el segundo valor (la posición 1 porque empezamos a contar desde 0)
2. Asignarle un nuevo valor que lo reemplace y volver a imprimirlo

3. Mostrar el contenido de la lista usando un bucle *for*
4. Mostrar aquellas temperaturas que superaron el promedio

```
# Elemento 1 de la lista
print("2do elemento:", tempC[1])

# Reemplaza el elemento 1 con 100
tempC[1] = 100
print("2do elemento modificado:",tempC[1])

# Lista completa y calculo de promedio
print("Lista:")
media = 0.0
for t in tempC:
    print(t)
    media = media + t
media = media/3

# Elementos que superan el promedio
for t in tempC:
    if t > media:
        print("La temperatura", t, "superó la media")
```

```
2do elemento: 33.3
2do elemento modificado: 100
Lista:
12.2
100
12.1
La temperatura 100 superó la media
```

Como se observa, las listas son fácilmente iterables utilizando el ciclo *for*, ya que al igual que una cadena de caracteres, es una secuencia de valores, la diferencia radica que en una cadena los valores son únicamente caracteres mientras que en una lista pueden ser de cualquier otro tipo, **incluso otra lista**. Veamos una lista que combine elementos de distintos tipos:

```
# Lista que almacena distintos tipos de datos
popurri = [12, 3.1415, "amapola del 66", True, tempC]

# Imprimen los elementos
print("1er elemento: ", popurri[0])
print("2do elemento: ", popurri[1])
print("3er elemento: ", popurri[2])
print("4to elemento: ", popurri[3])
print("5to elemento: ", popurri[4])
```

```
1er elemento:  12
2do elemento:  3.1415
3er elemento:  amapola del 66
```

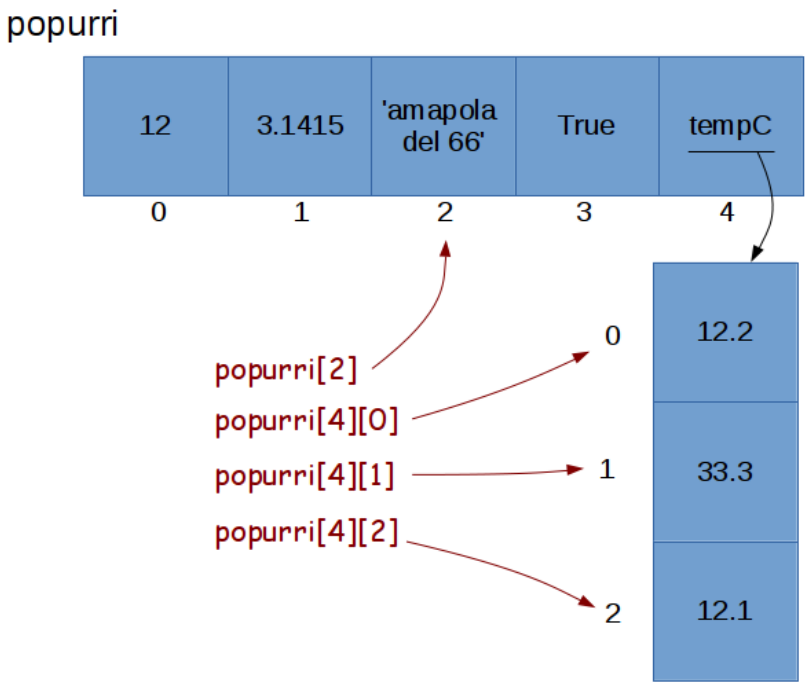
```
4to elemento:  True
5to elemento:  [12.2, 100, 12.1]
```

Ahora bien, seguramente el lector estará intrigado sobre el acceso a un elemento en particular de la lista *tempC*, ubicada en la 5ta posición de la lista *popurri*. En *popurri[4]* se referencia el elemento en cuestión, que es una lista, por lo que agregando un índice más accedemos a cada uno de sus elementos, veamos el código:

```
print(popurri[4][0])
print(popurri[4][1])
print(popurri[4][2])
```

```
12.2
100
12.1
```

En la siguiente figura se observa la estructura de esta lista.




*Lista con elementos de distintos tipos*

Una de las funcionalidades que nos provee Python para obtener información sobre la cantidad de elementos de las listas es *len()*. Veamos los resultados que arroja aplicado a la lista *popurri*.

```
print(len(popurri))
print(len(popurri[4]))
```

```
5
3
```

Otra alternativa para iterar sobre una lista es combinando la función *range* que vimos anteriormente y la cantidad de elementos de la lista, de manera que podemos acceder a los elementos a partir de su índice:

```
n = len(tempC)
for i in range(n):
    print("Temperatura", i, ":",tempC[i])
```

```
Temperatura 0 : 12.2
Temperatura 1 : 33.3
Temperatura 2 : 12.1
```

La función *len()* retornó la cantidad de elementos de la lista *tempC*, ese resultado, almacenado en *n*, fue utilizado como el valor para la función *range()* que generó una secuencia numérica (una lista!!!) que va desde 0 hasta *n-1*.

En el caso previo mostramos la posición de cada elemento, pero es posible iterar sobre la lista de una manera mucho mas directa:

```
for t in tempC:
    print("Temperatura:",t)
```

```
Temperatura: 12.2
Temperatura: 33.3
Temperatura: 12.1
```

En este caso, cada elemento de la lista se asigna a la variable *t* en forma sucesiva, de manera que en cada iteración *t* contendrá uno a uno los elementos de la lista *tempC*.

Las listas permiten **agregar elementos nuevos** en tiempo de ejecución utilizando el método *append()*. Veamos un ejemplo donde el usuario ingresa 10 nombres para agregar en una lista:

```
nombres = []            # lista inicialmente vacia
for i in range(10):
    un_nombre = input("Nombre: ")
    nombres.append(un_nombre)
```

***Note***

El término secuencia es el nombre genérico utilizado para toda estructura de datos que permita acceder a sus elementos a través de un índice comenzando en cero (por ej. nombres[0]) y con un tamaño conocido (*len(nombres)*). Esta denominación es común en Python desde sus comienzos y aplica para listas, tuplas, cadenas de caracteres, etc.

### *2.1.1 Listas bidimensionales*

Una lista bidimensional puede ser vista como una matriz, es decir, cada elemento se encuentra almacenado en una determinada fila y columna, por lo que para accederlo son necesarios dos índices. Veamos un caso de una lista bidimensional de tres filas y cinco columnas (3x5)

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| **0** | 12.2 | 33.3 | 12.1 | 0.3 | 1.21 |
| **1** | 3.14 | 2.1 | 9.8 | 28.1 | 19.8 |
| **2** | 10.8 | 0.1 | 0.2 | 22.1 | 9.38 |

Veamos el modo de definirla:

```
matriz = [
    [12.2, 33.3, 12.1, 0.3, 1.21],
    [3.14, 2.1, 9.8, 28.1, 19.9],
    [10.8, 0.1, 0.2, 22.1, 9.38]
    ]
```

El acceso a cada dato se realiza utilizando los dos índices, donde el primero hace referencia a la fila y el segundo a la columna. Así, si se accede al segundo elemento (columna 1) de la tercer fila (fila 2): *matriz[2][1]*.

El recorrido de una matriz se simplifica utilizando ciclos repetitivos anidados, veamos un posible modo de iterar por las columnas de la matriz previamente definida.

```
for c in range(5):
    print("Columna",c)
    for f in range(3):
        print(matriz[f][c])
    print()
```

```
Columna 0
12.2
3.14
10.8

Columna 1
33.3
2.1
0.1

Columna 2
12.1
9.8
0.2

Columna 3
0.3
28.1
```

```
22.1

Columna 4
1.21
19.9
9.38
```

Otro modo de recorrer una matriz es iterando directamente sobre sus elementos es:

```
for fila in matriz:
    for elemento in fila:
        print(elemento, ' ', end='')
    print()
```

En este caso, el *for* externo asigna en cada ciclo una fila diferente y el *for* anidado itera sobre cada elemento de esa fila. Veamos la salida que produce:

```
12.2  33.3  12.1  0.3  1.21
3.14  2.1  9.8  28.1  19.9
10.8  0.1  0.2  22.1  9.38
```

### *2.1.2 Operaciones*

En Python, las listas, las tuplas y las cadenas de caracteres son parte del conjunto de las secuencias. Todas las secuencias cuentan con las siguientes operaciones:

| Operación | Resultado |
|---|---|
| *x in s* | Indica si la variable x se encuentra en s |
| *s + t* | Concantena las secuencias s y t. |
| *s * n* | Concatena n copias de s. |
| *s[i]* | Elemento i de s, empezando por 0. |
| *s[i:j]* | Porción de la secuencia s desde i hasta j (no inclusive). |
| *s[i:j:k*] | Porción de la secuencia s desde i hasta j (no inclusive), con paso k. |
| *len(s)* | Cantidad de elementos de la secuencia s. |
| *min(s)* | Mínimo elemento de la secuencia s. |
| *max(s)* | Máximo elemento de la secuencia s. |

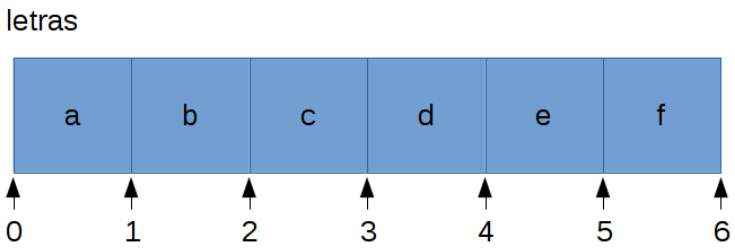
### *2.1.3 Rebanadas (slices)*

Para acceder a los elementos de una lista se puede usar como índice cualquier expresión entera, por lo que *tempC[1+1]* o *matriz[2*0+1][2*2]* son operaciones perfectamente válidas.

Además, existen expresiones que permiten extraer o modificar rebanadas o recortes de un conjunto de elementos de la lista . Veamos unos ejemplos.

```
>>> letras = ['a', 'b', 'c', 'd', 'e', 'f']
>>> letras[1:3]
['b', 'c']
>>> letras[:4]
['a', 'b', 'c', 'd']
>>> letras[3:]
['d', 'e', 'f']
>>> letras[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Una manera de visualizar más fácilmente una rebanada es pensar que los índices de las listas corresponden al límite de cada elemento, como se observa en el siguiente diagrama:




Siguiendo ese concepto, podemos reemplazar varios elementos a la vez:

```
>>> letras = ['a', 'b', 'c', 'd', 'e', 'f']
>>> letras[1:3] = ['x', 'y']
>>> print(letras)
['a', 'x', 'y', 'd', 'e', 'f']
```

Además, se pueden eliminar varios elementos asignándoles la lista vacía:

```
>>> letras = ['a', 'b', 'c', 'd', 'e', 'f']
>>> letras[1:3] = []
>>> letras
['a', 'd', 'e', 'f']
```

También es posible añadir elementos insertándolos en una porción vacía, en la posición deseada:

```
>>> letras = ['a', 'd', 'f']
>>> letras[1:1] = ['b', 'c']
>>> print(letras)
['a', 'b', 'c', 'd', 'f']
>>> letras[4:4] = ['e']
>>> print(letras)
['a', 'b', 'c', 'd', 'e', 'f']
```

***Note***

El *slicing* es una cualidad común de todas las secuencias en Python. La razón de la exclusión del último valor tiene sus ventajas:

- Es fácil de ver la longitud cuando solamente se indica la posición de final, *letras[:4]*, tiene 4 elementos.
- Es fácil de calcular la longitud cuando se da el rango con el inicio y fin, *letras[1:3]*, tiene 2 (fin-inicio) elementos.
- Es fácil dividir una secuencia en dos partes sin solape, *letras[:4]* y *letras[4:]* nos da la secuencia completa de letras.

#### *2.1.4 Métodos*

Una lista provee una serie de funcionalidades asociadas denominados métodos. Se propone profundizar sobre los métodos disponibles con la lectura del *Tutorial de Python* (pág. 26, *Más sobre listas*)

- *list.append(x)* Agrega un ítem al final de la lista. Equivale a *list[len(list):] = [x]*
- *list.extend(L)* Extiende la lista agregándole todos los ítems de la lista dada. Equivale a *list[len(list):] = L*
- *list.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 *list.insert(0, x)* inserta al principio de la lista, y *list.insert(len(list),x) equivale a list.append(x)*
- *list.remove(x)* Quita el primer ítem de la lista cuyo valor sea x. Es un error si no existe tal ítem
- *list.pop([,i])* Quita el ítem en la posición dada de la lista, y lo devuelve. Si no se especifica un índice a.pop() quita y devuelve el último ítem de la lista. (Los corchetes que encierran a i en la firma del método denotan que el parámetro es opcional, no que deberías escribir corchetes en esa posición. Verás esta notación con frecuencia en la Referencia de la Biblioteca de Python.)
- *list.clear()* Quita todos los elementos de la lista. Equivalente a *del list[:]*
- *list.index(x)* Devuelve el índice en la lista del primer ítem cuyo valor sea x. Es un error si no existe tal ítem
- *list.count(x)* Devuelve el número de veces que x aparece en la lista
- *list.sort()* Ordena los ítems de la lista in situ
- *list.reverse()* Invierte los elementos de la lista in situ
- *list.copy()* Devuelve una copia superficial de la lista. Equivalente a *list[:]*

Una manera de quitar un ítem de una lista dado su índice en lugar de su valor es la instrucción *del*, que también puede usarse para quitar secciones de una lista o vaciar la lista completa. Por ejemplo:

```
a = [-1, 1, 66.25, 333, 333, 1234.5]
del a[0]
a
[1, 66.25, 333, 333, 1234.5]
del a[2:4]
a
[1, 66.25, 1234.5]
```

#### *2.1.5 Listas por comprensión*

Una forma rápida de construir una secuencia (en este caso una lista) es utilizando una lista por comprensión (en inglés *list comprehensions* o en forma corta *list-comps*). En un ejemplo previo agregamos diez nombres en una lista utilizando el siguiente código:

```
nombres = []           # lista inicialmente vacia
for i in range(10):
    un_nombre = input("Nombre: ")
    nombres.append(un_nombre)
```

En este caso, utilizamos cuatro líneas de código, sin embargo, usando *list-comps* se podría lograr el mismo efecto reduciendo el código a lo siguiente:

```
nombres = [input("Nombre: ") for i in range(10)]
```

Ahora veamos un ejemplo donde generamos una nueva lista con aquellos nombres que tienen la letra "n". Para este caso debemos incorporar un *if "n" in nombre*:

```
nombres_con_n = [nom for nom in nombres if "n" in nom]
```

> ***Note***
>
> Las listas son un tipo flexible y fácil de usar pero dependiendo de los requerimientos existen mejores opciones. Si se necesita almacenar millones de valores de punto flotante una lista no es la mejor alternativa, para esto existe el tipo *array* de la biblioteca estándar o la biblioteca externa *NumPy*, que maneja arreglos numéricos sumamente eficientes.

## 2.2 Tuplas

Las tuplas son secuencias, al igual que las listas. Generalmente son descriptas como listas inmutables, es decir, una vez creadas no pueden ser modificadas (al igual que las cadenas de caracteres).

Una tupla consiste de un número de valores separados por comas, que pueden ingresarse con o sin paréntesis, por ejemplo:

```
tu = 28, 21, 'hola!'                          # sin paréntesis
coordenadas = (-31.6251956,-60.7108061)       # con paréntesis
print(tu[0])
print(tu)
print(coordenadas)

28
(28, 21, 'hola!')
(-31.6251956,-60.7108061)
```

El desempaquetado consiste en asignar en variables a cada elemento de la secuencia, veamos el siguiente ejemplo:

```
latitud, longitud = coordenadas
```

Además es posible hacer un desempaquetado selectivo, asignando solamente aquellos elementos de la secuencia que nos interesen, por ejemplo para una tupla *fecha = ("28","marzo","1981")* desempaquetamos almacenando únicamente el último valor en una variable *anio*

```
_,_, anio = fecha
```

Es importante destacar que el desempaquetado no es exclusivo de las tuplas sino que es propio de todas las estructuras de datos que son secuencias.

Para mayor detalle sobre esta estructura se recomienda leer el Tutorial de Python, *Tuplas y secuencias*, pag. 31.

## 2.3 Diccionarios

Hemos visto que las listas son útiles cuando se quiere agrupar datos en una estructura y acceder a cada uno de ellos a través de su índice.

Otro tipo de estructura que nos permite referirnos a un determinado valor a través de un nombre o clave es un diccionario. Muchas veces este tipo de estructura es más apropiado que una lista.

El nombre *diccionario* da una idea sobre el propósito de la estructura ya que uno puede realizar fácilmente una búsqueda a partir de una palabra específica (*clave*) para obtener su definición (*valor*).

Un ejemplo podría ser una agenda telefónica, que nos permita obtener el número de teléfono de una persona a partir de su nombre. Veamos entonces el modo de crear diccionarios.

```
agenda = {'Marado':'1552123', 'Dolina':'4584129',
          'Spasiuk':'65748', 'Fontanarrosa':'32456'}
```

El acceso a un valor se realiza a partir de su clave, por ejemplo:

```
print(agenda['Marado'])
print(agenda['Fontanarrosa'])
```

```
1552123
32456
```

Los *diccionarios* consisten en pares llamados *ítems* formados por *claves* y sus *valores* correspondientes. En este ejemplo, los nombres son las claves y los números de teléfono son los valores. Cada clave es separada de su valor por los dos puntos (:), los ítems son separados por comas, y toda la estructura es encerrada entre llaves. Un diccionario vacío, sin ítems, se escribe con solo dos llaves: *{}*.

Las claves, debido a que funcionan como índices, no pueden ser repetidas. Veamos las formas más comunes de iterar sobre un diccionario:

#### *2.3.1 Iterar sobre las claves*

```
for nom in agenda:
    print(nom)

Spasiuk
Marado
Dolina
Fontanarrosa
```

#### *2.3.2 Iterar sobre los valores*

```
for tel in agenda.values():
    print(tel)

65748
1552123
4584129
32456
```

#### *2.3.3 Iterar sobre los items*

```
for nom, tel in agenda.items():
    print(nom, tel)

Spasiuk 65748
Marado 1552123
Dolina 4584129
Fontanarrosa 32456
```

Al igual que las listas, los diccionarios son sumamente flexibles y pueden estar formados por otros diccionarios (o inclusive listas). Analicemos un breve ejemplo de un diccionario que está conformado del siguiente modo:

- Cuenta con tres ítems

- El valor de cada ítem es otro diccionario que a su vez contiene tres ítems con las claves *titulo*, *fecha* y *autor*

A continuación veamos la implementación de esta estructura, la impresión manual y mediante iteración:

```
referencia = { "libro1":{"titulo":"El tutorial de Python",
                         "fecha":"2013",
                         "autor":"Guido van Rossum"},
               "libro2":{"titulo":"Aprenda a Pensar Como un \
                            Programador con Python",
                         "fecha":"2002",
                         "autor":"Allen Downey"},
               "libro3":{"titulo":"Inmersión en Python 3",
                         "fecha":"2009",
                         "autor":"Mark Pilgrim"}
             }
# acceso a los valores de titulo de cada libro
print("Títulos")
print("=======")
print(referencia["libro1"]["titulo"])
print(referencia["libro2"]["titulo"])
print(referencia["libro3"]["titulo"])
print()

# Mezcladito
for clave in referencia:
    print(clave)
    print("======")
    for clave2, val in referencia[clave].items():
        print(clave2, val, sep=": ")
    print()
```

```
Títulos
=======
El tutorial de Python
Aprenda a Pensar Como un Programador con Python
Inmersión en Python 3

libro3
======
autor: Mark Pilgrim
titulo: Inmersión en Python 3
fecha: 2009

libro2
======
autor: Allen Downey
titulo: Aprenda a Pensar Como un Programador con Python
fecha: 2002
```

```
libro1
======
autor: Guido van Rossum
titulo: El tutorial de Python
fecha: 2013
```

#### *2.3.4 Operaciones*

- *len(d)* retorna el número de items (pares clave-valor) en d
- *d[k]* retorna el valor asociado con la clave k
- *d[k] = v* asocia el valor v con la clave k
- *del d[k]* elimina el item con clave k
- *k in d* evalúa si existe un item en d que tenga la clave k

Aunque las listas y los diccionarios comparten varias características en común, existen ciertas distinciones importantes:

- Tipos de claves: Las claves de los diccionarios no deben ser enteros (aunque pueden serlo). Deben ser tipos de datos inmutables (números flotantes, cadenas de caracteres o tuplas)
- Agregado automático: En un diccionario se crea un ítem automáticamente al asignar un valor a una clave inexistente, en una lista no se puede agregar un valor en un índice que esté fuera del rango.
- Contenido: La expresión *k in d* (d es un diccionario) evalúa por la existencia de una clave, no de un valor. Por otro lado, la expresión *v in l* (siendo l una lista), busca por un valor en vez de por un índice.

#### *2.3.5 Métodos*

A continuación se describen brevemente algunos de los métodos más utilizados:

- *clear()* Elimina todos los ítems
- *copy()* Retorna una copia superficial del diccionario
- *get(key[, default])* Retorna el valor de la clave key si existe, sino el valor default. Si no se proporciona un valor default, entonces retorna None.
- *items()* Retorna el par de valores del ítem clave, valor.
- *keys()* Retorna las claves.
- *pop(key[, default])* Si la clave key está presente en el diccionario la elimina y retorna su valor, sino retorna default. Si no se proporciona un valor default y la clave no existe se produce un error (KeyError).
- *popitem()* Elimina y retorna un par (clave, valor) arbitrario.
- *setdefault(key[, default])* Si la clave key está presente en el diccionario retorna su valor. Si no, inserta la clave con un valor de default y retorna default

- *update([other])* Actualiza los ítems de un diccionario en otro. Es útil para concatenar diccionarios.
- *values()* Retorna los valores del diccionario.

Los diccionarios pueden ser comparados por su igualdad si y solo si tienen los mismos ítems. Otras comparaciones (<, <=, >=, >) no son permitidas.

#### 2.3.6 Igualdad entre diccionarios

¿Cuándo un diccionario es igual a otro? Aclaremos esta intrigante pregunta analizando el siguiente resultado:

```
>>> a = dict(uno=1, dos=2, tres=3)
>>> b = {'uno': 1, 'dos': 2, 'tres': 3}
>>> c = dict([('dos', 2), ('uno', 1), ('tres', 3)])
>>> d = dict({'tres': 3, 'uno': 1, 'dos': 2})
>>> a == b == c == d
True
```

## 2.4 Conversión entre listas y diccionarios

#### 2.4.1 De diccionarios a listas

Es posible crear listas a partir de diccionarios usando los métodos *items()*, *keys()* y *values()*. El método *keys()* crea una lista que consiste solamente en las claves del diccionario, mientras que *values()* produce una lista que contiene los valores. *items()* puede ser usado para crear una lista que conste de tuplas de dos pares (clave, valor). Utilicemos el diccionario agenda creado anteriormente:

```
print("Lista de ítems")
print("==============")
items_vista = agenda.items()
items = list(items_vista)
print(items)
print()

print("Lista de claves")
print("===============")
claves_vista = agenda.keys()
nombres = list(claves_vista)
print(nombres)
print()

print("Lista de valores")
print("===============")
valores_vista = agenda.values()
telefonos = list(valores_vista)
print(telefonos)
```

```
Lista de ítems
==============
[('Dolina', '4584129'), ('Fontanarrosa', '32456'),
('Spasiuk', '65748'), ('Marado', '1552123')]

Lista de claves
===============
['Dolina', 'Fontanarrosa', 'Spasiuk', 'Marado']

Lista de valores
================
['4584129', '32456', '65748', '1552123']
```

#### *2.4.2 De listas a diccionarios*

Ahora realizaremos el proceso inverso, para armar un diccionario a partir de dos listas. Ya en el ejemplo previo obtuvimos dos listas, una con los nombres y otra con los teléfonos. Las funciones a utilizar son 3: *zip()*, *list()* y *dict()*. Veamos:

```
lista_de_tuplas = list(zip(nombres, telefonos))
agenda2 = dict(lista_de_tuplas)
print(agenda2)
```

```
{'Fontanarrosa': '32456', 'Dolina': '4584129',
 'Spasiuk': '65748', 'Marado': '1552123'}
```

## 2.5 Cadenas de caracteres

Una cadena es una secuencia de caracteres. Hasta aquí solamente las utilizamos para mostrar mensajes pero sus usos son mucho más amplios, a continuación las veremos en mayor detalle.

Es importante destacar:

- Las cadenas son inmutables: una vez creadas no podemos modificarlas accediendo manualmente a sus caracteres.
- El acceso a sus caracteres es igual al de los elementos de una lista: el primer caracter se encuentra en la posición cero y se puede acceder a sus caracteres utilizando rebanadas o porciones (slices).

Veamos la siguiente cadena:

```
frase = 'siento que nací en el viento'
```

- Obtenemos la cantidad de caracteres utilizando la función *len(frase)*
- Accedemos a los caracteres usando índices, por ejemplo, el cuarto caracter se encuentra en *frase[3]*
- Soporta rebanadas, podemos extraer por ejemplo la segunda palabra, *frase[7:10]*
- La última palabra: *frase[-6:]*

### *2.5.1 Operaciones*

Hemos visto ya dos operadores matemáticos que son compatibles para su uso con cadenas de caracteres: operador suma (+) y el multiplicación (*). Recordemos su funcionamiento con un simple ejemplo

```
w = "libertad"
print(3*(w+' '))
```

```
libertad libertad libertad
```

Las cadenas de caracteres pueden ser comparadas mediante los símbolos: >, >=, <, <=, ==, !=. Veamos un ejemplo:

```
palabra = input("Ingrese una palabra: ")
if palabra < w:
    print("Tu palabra, "+palabra+ ", va antes que " + w)
elif palabra > w:
    print("Tu palabra, "+palabra+ ", va después que " + w)
else:
    print("Tu palabra, "+palabra+ ", es " + w)
```

```
Ingrese una palabra: cadenas
Tu palabra, cadenas, va antes que libertad
```

### *2.5.2 Métodos*

Las cadenas también cuentan con métodos que realizan una función específica, a continuación vemos los más usuales:

- *find* Busca una subcadena dentro de otra.
- *lower* y *upper* Retorna la cadena en minúsculas y mayúsculas respectivamente.
- *replace* Retorna una cadena donde todas las ocurrencias de una cadena son reemplazadas por otra.
- *split* Separa una cadena según un caracter separador y retorna una lista con los elementos separados.
- *strip* Retorna una cadena donde los espacios en blanco al inicio y al final de la cadena son eliminados, pero no los interiores.
- *join* Es el inverso de *split*. Une elementos de una lista en una cadena de caracteres usando un caracter de separación.

Apliquemos algunos de estos métodos:

```
print(frase.find("nací"))
print(frase.lower())
print(frase.upper())
print(frase.replace("viento", "hospital"))
lista_frase = frase.split(" ")
print(lista_frase)
sep = "-"
print(sep.join(lista_frase))
```

```
11
siento que nací en el viento
SIENTO QUE NACÍ EN EL VIENTO
siento que nací en el hospital
['siento', 'que', 'nací', 'en', 'el', 'viento']
siento-que-nací-en-el-viento
```