UNL VIRTUAL

Tecnicatura Universitaria en Software Libre

Introducción al Desarrollo de Software

Unidad N° 3
Estructuras de datos y control de flujo

Autor
Emiliano López

Colaborador
Maximiliano Boscovich

FICH UNL

Introducción al desarrollo de software

Autor: Emiliano López - elopez@fich.unl.edu.ar
Colaborador: Maximiliano Boscovich - maximiliano@boscovich.com.ar
Fecha: 14/07/2017 16:33 - [última versión disponible]

Unidad III: Estructuras de datos y control de flujo

1 Control de flujo

Con lo aprendido hasta la unidad previa, la ejecución de un programa no es más que una lista de órdenes a ejecutar de forma **secuencial**. Independientemente de los datos de entrada el camino del programa es indefectiblemente el mismo.

Se ejecutarán siempre las mismas instrucciones, en forma secuencial, una tras otra. Esta limitante quita flexibilidad a los programas ya que no es posible tener caminos alternativos de ejecución y, cada instrucción se ejecuta una única vez.

Pensemos en un ejemplo muy simple, donde se deben ingresar miles de datos de personas, sería impracticable incluir miles de sentencias para leer su nombre y edad.

Con este ejemplo sencillo, vemos la necesidad de contar con algo más que la ejecución secuencial aprendida previamente, por lo que para solucionar esta limitación existen las estructuras de control de flujo que permiten por un lado **condicionar** las acciones a ejecutarse y, por el otro, **repetir** una serie de instrucciones.

El teorema de la programación estructurada nos dice que todo algoritmo computacional puede ser resuelto utilizando tres estructuras:

- Secuencial
- Condicional
- Repetitiva

En la presente unidad, agregaremos a la ya vista estructura secuencial, estructuras condicionales y repetitivas.

1.1 Estructuras condicionales

La primer estructura de control que veremos son los condicionales. Su función principal es evaluar ciertas condiciones y en base al resultado llevar a cabo la ejecución de un fragmento de programa u otro. Aquí es donde cobra importancia el tipo lógico que aprendimos en la sección anterior (Unidad 2: Tipos básicos) ya que el resultado de las condiciones a evaluar pueden ser Verdadero (*True*) o Falso (*False*).

1.1.1 Sentencia if

La forma más simple de un estamento condicional es un *if* (*si*) seguido de la condición a evaluar y dos puntos (:). A partir de la siguiente línea se escribe el código a ejecutar en caso que se cumpla dicha condición (su resultado sea *True*), indicando este bloque de sentencias con una sangría.

```
if condicion:
    accion1    # bloque de
    ...        # sentencias
    accionN    # a ejecutar
```

Pensemos en un programa que hace ciertas preguntas y en base a las respuestas nos informe si conviene ir al trabajo en bicicleta o en auto. Este programa podría considerar la temperatura, la hora y la distancia y en base a estas variables tener un comportamiento diferenciado.

Iniciemos con el caso más simple, teniendo en cuenta solamente la temperatura para decidir el camino del programa:

```
temperatura = 12
if (temperatura > 10) and (temperatura < 30):
    print('Está lindo para bici!')
```

```
Está lindo para bici!
```

Esta sentencia se lee: Si la temperatura es mayor a 10 y también menor a 30 imprimir en pantalla *Está lindo para bici!*. Este mensaje se mostrará solamente al cumplirse la condición, es decir, cuando la variable temperatura contenga un valor entre 10 y 30. En otro caso, el programa no mostrará nada.

Una característica saliente de Python para este tipo de comparaciones es la de asemejarse al lenguaje natural (en inglés) y soportar comparaciones similares al lenguaje matemático por lo que podemos implementar una forma equivalente a la comparación previa haciendo:

```
if 10 < temperatura < 30:
    print('Está lindo para bici!')
```

Todo lenguaje de programación tiene en su sintaxis un modo de identificar las acciones que forman parte de un bloque, por ejemplo, en C++ y Java se utilizan llaves para encerrar las sentencias que se deben ejecutar en caso que el resultado de la comparación sea verdadero, aquí, en Python, se **utiliza la sangría**.

Es importante indentar el bloque de acciones tal como se ha hecho en el ejemplo, es decir, dejar una sangría en las líneas debajo de los dos puntos (:) para indicar todas aquellas instrucciones que se deben ejecutar en caso que la condición evaluada sea verdadera.

Si quisiéramos mostrar varios mensajes sería del siguiente modo:

```
if 10 < temperatura < 30:
    print('Está lindo para bici!')
    print('Pedalear un rato hace bien!')
```

1.1.2 Sentencia if..else

El problema inicialmente planteado consiste en determinar el modo de ir al trabajo, en vehículo o bicicleta, sin embargo el programa imprime en pantalla solamente cuando sugiere ir en bici y, en caso que la condición fuera falsa, no se muestra mensaje alguno.

Lo novedoso es que se agregó una posibilidad de no ejecución de ciertas acciones, sin embargo, para completar este problema es necesario que existan dos caminos alternativos de ejecución, uno para la condición verdadera y otro para cuando sea falsa, de este modo, algunas de las instrucciones se ejecutará, pero no ambas.

Para estos casos existe la sentencia *else* (*sino*), que se usa conjuntamente con *if* y que sirve para ejecutar ciertas instrucciones en caso de que la condición de la evaluada no se cumpla. Completando el ejemplo:

```
if 10 < temperatura < 30:
    print('Está lindo para ir en bici')
else:
    print('Te recomiendo ir en cole')
```

Esto se lee como *si temperatura es mayor a 10 y además menor que 30, entonces mostrar el mensaje 'Está lindo para ir en bici', sino es así, mostrar el mensaje 'Te recomiendo ir en cole'*. Siempre se ejecutará una de las dos opciones, dependiendo del valor de la variable temperatura. Por lo que en este punto podemos decir que el código se bifurca en dos caminos diferentes dependiendo de una condición.

En este caso también tenemos que prestar atención a la indentación utilizada bajo la sentencia *else*, se escribe al mismo nivel que la sentencia *if*.

Una versión más completa del programa podría ser la siguiente:

```
temperatura = int(input('Ingrese la temperatura en ºC: '))

if 10 < temperatura < 30:
    print('Está lindo para ir en bici')
else:
    print('Te recomiendo ir en cole')
print('Que tenga buen día!')
```

Es importante mencionar que la última sentencia siempre se ejecutará, la bifurcación se produce solamente entre las sentencias que están dentro del *if* y el *else*, el mensaje 'Que tenga buen día!' se mostrará independientemente del camino que haya tomado la ejecución del programa.

1.1.3 Estructura de selección múltiple if..elif..else

En los casos previos la secuencia de ejecución del programa tiene solamente dos alternativas, el bloque de acciones cuando la condición es verdadera (*True*) o cuando es falsa (*False*), incluso, tal como se planteó en el primer ejemplo, puede no existir un camino por la alternativa falsa.

Las estructuras de selección múltiple sirven para evaluar mas de una condición y por ende posibilitar varios caminos de ejecución del programa. En Python, la forma es la siguiente:

```
if condicion1:
    acciones
    ...
elif condicion2:
    acciones
    ...
elif condicion3:
    acciones
    ...
```

```
else:
    acciones
    ...
```

La interpretación de esta sentencia significa que cuando cumpla alguna de las condiciones ingresará al bloque de acciones correspondientes y, en caso que no cumpla con ninguna, ejecutará las acciones del *else*, que podría ser omitido si no son necesarias acciones por defecto.

Veamos un ejemplo para mejorar la comprensión. Se lee una nota numérica de una evaluación (0..100) y el programa debe mostrar una calificación cualitativa según la siguiente escala:

- Insuficiente (nota < 60)
- Aprobado (60 <= nota < 70)
- Bueno (70 <= nota < 80)
- Muy Bueno (80 <= nota < 90)
- Distinguido (90 <= nota < 100)
- Sobresaliente (nota = 100)

```
# Lectura de la nota
nota = int(input('Ingrese la nota (0..100): '))
# Decide la calif. correspondiente
if nota < 60:
    calif = "Insuficiente"
elif 60 <= nota < 70:
    calif = "Aprobado"
elif 70 <= nota < 80:
    calif = "Bueno"
elif 80 <= nota < 90:
    calif = "Muy Bueno"
elif 90 <= nota < 100:
    calif = "Distinguido"
else:
    calif = "Sobresaliente"
# Mensaje alusivo
print("Calificación: ", calif)
```

Como se observa, cada expresión condicional planteada es excluyente de las demás, por lo que no puede cumplir con mas de una a la vez. Ahora, podría existir un planteo donde se cumplan más de una condición y la pregunta obvia es, ¿qué sucede en ese caso?

Analicemos el siguiente programa, ¿qué mensaje se muestra en pantalla?

```
val = 85
if val > 81:
    print("opción 1")
elif val > 82:
    print("opción 2")
```

```
elif val > 83:
    print("opción 3")
```

1.1.4 Estructuras anidadas

Retomando el ejemplo del programa anterior, consideremos además de la temperatura la distancia que se debe recorrer. Para estos casos, se pueden utilizar *estructuras anidadas*, es decir, una nueva estructura de control incluida dentro del bloque que se ejecuta al cumplirse la primer condición.

Reescribamos el código previo utilizando estructuras anidadas:

```
temperatura = int(input('Ingrese la temperatura en ºC: '))
distancia = int(input('Ingrese la distancia a recorrer en km: '))

if 10 < temperatura < 30:   #1er condicional
    if distancia <= 15:     #2do condicional
        print('Lindo clima para ir en bici')
    else:
        print('Es lejos, te recomiendo cole')
else:
    print('No está agradable, recomiendo cole')

print('Que tenga buen día!')
```

```
Ingrese la temperatura en ºC: 15
Ingrese la distancia a recorrer en km: 1
Está lindo para ir en bici
Que tenga buen día!
```

En caso de cumplirse el primer condicional pasa a considerarse el valor de la variable distancia con el segundo condicional, mostrando en pantalla *Lindo clima para ir en bici* si el resultado es verdadero y, *Es lejos, te recomiendo cole*, si es falso.

Por otro lado, si el primer condicional no se cumple (la temperatura no esta entre 10 y 29) se muestra el mensaje *No está agradable, recomiendo cole*.

La última sentencia, mostrará el mensaje *Que tenga buen día!* independientemente del valor de las variables *temperatura* y *distancia*.

1.2 Estructuras repetitivas

Ahora podemos dotar a nuestros programas de mayor complejidad, combinando y anidando las estructuras condicionales vistas. Sin embargo, aún tenemos una limitante, cada instrucción tendrá vida al momento de su ejecución y no se ejecutará más hasta que se el programa se invoque nuevamente.

Imaginemos que debemos consultar la pregunta de la temperatura a cientos de miles de personas, deberíamos ejecutar cientos de miles de veces el programa, iniciándolo y esperando su

finalización para repetir el proceso una y otra vez, o bien, copiando cientos de miles de veces el código del programa.

Con este inconveniente se hace evidente la necesidad de una estructura que permita repetir cuantas veces se requiera una determinada instrucción o bloque de instrucciones, aquí es donde entran en acción las estructuras repetitivas.

1.2.1 Sentencia while

El *while* permite repetir una serie de acciones **mientras** que una determinada expresión (o condición) se cumpla, en caso contrario, se finaliza la repetición.

Una expresión se cumple cuando arroja un resultado verdadero, que en Python es *True*. La forma genérica del *while* es la siguiente:

```
while <expresion>
    accion1
    accion2
    ...
    accionN
```

Tal como se explicó previamente, las acciones que se repiten en cada iteración son aquellas que tienen sangría, lo que indica que son parte del ciclo *while*.

En función del modo en que se controla la cantidad de repeticiones del ciclo se los clasifica en bucles condicionales, interactivos o centinelas.

1.2.2 Bucles condicionales

Veamos un ejemplo donde se pregunte el valor de temperatura a cinco personas y sugiera ir caminando si el clima es agradable (mayor a 16 °C) o en caso contrario en vehículo. Tomemos una estrategia para resolver el problema en tres pasos:

1. Leemos una temperatura que se ingresa por teclado
2. **Escribimos en pantalla un mensaje según la condición planteada:**
    - *Ir caminando* si la temperatura es mayor a 16 °C.
    - *Ir en vehículo* en caso contrario.
3. Repetir los dos pasos previos un total de cinco veces

Pasos 1 y 2

```
temperatura = int(input('Ingrese la temperatura en ºC:'))
if (temperatura > 16):
    print('Vas caminando')
else:
    print('Mucho frío, en vehículo')
```

Paso 3

Debemos incluir los pasos previos en una estructura que repita 5 veces. Pensemos lo anterior como un único bloque denominado *Pasos1y2*, y una manera de controlar cinco repeticiones. Para esto, usamos una variable con un valor inicial conocido que incrementamos en una unidad luego de cada ejecución del bloque que denominamos *Pasos1y2*. La estructura de nuestro programa podría ser la siguiente:

```
vez = 1              # valor inicial conocido
while vez <= 5:      # condicional para repetir
    Pasos1y2         # bloque Pasos1y2
    vez = vez + 1    # incremento
```

Al finalizar la ejecución de la instrucción *vez = vez + 1* la estructura iterativa evalúa nuevamente la expresión *vez <= 5* cuyo resultado puede ser cierto o falso (*True* o *False*).

Si el resultado es *True*, entonces el ciclo continuará con las acciones contenidas, re-evaluando la expresión en cada iteración y finalizando cuando sea *False*, es decir, cuando la variable *vez* ya no sea menor o igual que 5.

Ahora que ya hemos desmenuzado el inofensivo código previo, podemos pasar a la versión final del programa y ver su comportamiento.

```
vez = 1
while vez <= 5:
    temperatura = int(input('Ingrese la temperatura en ºC:'))
    if (temperatura > 16):
        print('Vas caminando')
    else:
        print('Mucho frío, en vehículo')
    vez = vez + 1
```

```
Ingrese la temperatura en ºC:12
Mucho frío, en vehículo
Ingrese la temperatura en ºC:16
Mucho frío, en vehículo
Ingrese la temperatura en ºC:17
Vas caminando
Ingrese la temperatura en ºC:18
Vas caminando
Ingrese la temperatura en ºC:20
Vas caminando
```

Este tipo de ciclo repetitivo, donde la cantidad de iteraciones depende de una condición es denominado **bucles condicionales** y cuenta con dos características destacables:

- El valor a ser evaluado en la expresión debe estar previamente definido
- En cada iteración el valor a ser evaluado en la expresión debe modificarse

Lo referido en el primer ítem evita obtener un mensaje de error, ya que no es posible evaluar una expresión con una variable que aún no fue definida, es decir, que no tiene asignado valor alguno.

La segunda característica evita tener un **bucle infinito** y por ende un programa que nunca finalice. Este tipo de errores es más difícil de detectar, ya que a priori el ejemplo parecería correcto.

1.2.3 Bucles interactivos

El ciclo *while* se adapta fácilmente para aquellos casos donde la repetición depende de un valor que ingresa el usuario, es decir, para aquellos programas donde la condición de corte o de repetición sea interactiva. Veamos un ejemplo en el que se calcula el promedio de valores numéricos ingresados por el usuario.

Pensemos una posible estrategia para su solución: el programa solicitará un nuevo valor numérico mientras que el usuario responda *si* a una pregunta, a su vez sumará y contará los valores numéricos ingresados.

Veamos el pseudocódigo del algoritmo mencionado:

```
Inicializar variable suma para sumar los números
Inicializar variable cant para contar los números
Inicializar variable mas_datos para almacenar respuesta del usuario (si/no)
Mientras la variable mas_datos sea si:
    Leer en x el nuevo valor numérico
    Sumarlo a la variable suma
    Contarlo
    Preguntar al usuario si sigue ingresando números
Mostrar en pantalla el promedio
```

Ahora veamos lo directa que es la traducción del algoritmo al lenguaje Python y su ejecución:

```
suma = 0.0
cant = 0
mas_datos = 'si'
while mas_datos == 'si':
    x = int(input('Ingrese valor'))
    suma = suma + x
    cant = cant + 1
    mas_datos = input('¿Mas valores (si/no)?')
print('El promedio de valores es', suma/cant)
```

```
Ingrese valor12
¿Mas valores (si/no)?si
Ingrese valor3
¿Mas valores (si/no)?si
Ingrese valor44
¿Mas valores (si/no)?no
El promedio de valores es 19.666666666666668
```

La limitación que encontramos está dada por la incomodidad de tener que ingresar dos valores por ciclo, uno para el dato numérico y otro para controlar si el usuario desea continuar o no. En ciertos casos puede ser la única alternativa, sin embargo, en otros se puede utilizar los bucles centinelas que se describen a continuación.

1.2.4 Bucles centinelas

Los bucles centinelas son aquellos donde la condición de corte tiene que ver con un valor que se diferencia del patrón que se ingresará y, será útil para discernir el momento en que corresponda continuar o bien finalizar la repetición.

Para el caso del cálculo del promedio, suponiendo que todos los valores serán siempre positivos podríamos tomar la estrategia de controlar que el valor ingresado sea mayor a cero para continuar la iteración. El pseudocódigo, obviando los detalles, sería similar al siguiente:

```
Leer en x el primer valor numérico
Mientras el valor x no sea el centinela:
    Sumarlo a la variable suma
    Contarlo
    Leer en x el nuevo valor numérico
Mostrar en pantalla el promedio
```

Veamos la implementación del programa en Python:

```
suma = 0.0
cant = 0
x = int(input('Ingrese valor (negativo para salir)'))
while x > 0:
    suma = suma + x
    cant = cant + 1
    x = int(input('Ingrese valor (negativo para salir)'))
print('El promedio de valores es', suma/cant)
```

Se debe tener el cuidado de mantener exactamente el mismo mensaje previo a ingresar al ciclo y en la última instrucción, para dar al usuario una idea de continuidad viendo una y otra vez el mismo comportamiento.

En el ejemplo expuesto, la limitación surge cuando se requiera promediar valores negativos. Sin embargo, Python provee herramientas que permiten salvar este inconveniente.

Veamos la estrategia para una posible solución:

1. Solicitar al usuario ingrese el valor numérico o que presione *enter* para salir
2. Evaluar en la expresión de corte e iterar mientras el valor ingresado no sea vacío
3. Realizar los cálculos

Traduzcamos esta estrategia a código Python y veamos su comportamiento:

```
suma = 0.0
cant = 0
x = input('Ingrese valor (<enter> para salir)')
while x != '':
    suma = suma + eval(x)
    cant = cant + 1
    x = input('Ingrese valor (<enter> para salir)')
print('El promedio de valores es', suma/cant)
```

```
Ingrese valor (<enter> para salir)12
Ingrese valor (<enter> para salir)-2
Ingrese valor (<enter> para salir)-3
Ingrese valor (<enter> para salir)23
Ingrese valor (<enter> para salir)2
Ingrese valor (<enter> para salir)
El promedio de valores es 6.4
```

El valor leído en *x* no se convierte en un número entero, sino que se lo mantiene como *str* hasta el momento de sumarlo a la variable *suma* utilizando la función *eval()*. Cuando el usuario presione *enter* el caracter en *x* será vacío y no ingresará al ciclo *while*.

1.2.5 Bucles infinitos y break

En todos los casos previos la finalización de la repetición se da por la condición dada en el *while*, sin embargo, es muy común utilizar ciclos con una condición *True* constante y quebrar la iteración utilizando la sentencia *break*.

La estructura general de estos ciclos es del siguiente modo:

```
while True:
    acciones...
    if ALGUNA_CONDICION:
        break
```

Cuando se ejecuta la sentencia *break* se finaliza el ciclo iterativo que lo contiene. La ventaja de este ciclo respecto a los anteriores es que permite ejecutar sentencias antes del corte del ciclo. En el ejemplo siguiente tenemos la estructura de un juego simplificado donde la finalización se da ante dos condiciones, al ganar o al perder, y previo al break se informa con un mensaje alusivo.

```
while True:
    acciones...
    if CONDICION_GANA:
        print("Felicitaciones, ganaste!")
        break

    if CONDICION_PIERDE:
        print("Perdiste!!")
        break
```

1.2.6 Sentencia for

La sentencia *for* provee otro modo de realizar bucles repetitivos en la mayoría de los lenguajes de programación y por supuesto en Python. Si bien la elección de un bucle u otro muchas veces dependerá del gusto del programador, para ciertos casos suele ser más cómoda una estructura que otra.

Veamos la sintaxis básica del bucle for:

```
for <var> in <secuencia>:
    accion1
    accion2
    ...
    accionN
```

El *for* ejecuta el bloque de acciones tantas veces como elementos contenga la *secuencia*, y en cada iteración la variable *var* almacenará uno a uno sus valores.

El significado de secuencia para Python puede variar desde cadenas de caracteres a listas de valores, en forma simplificada podemos definir una secuencia como toda *estructura de datos formada por elementos por los que se puede iterar*.

Veamos un ejemplo donde mostramos los caracteres de una cadena.

```
palabra = 'estimados'
for letra in palabra:
    print(letra)
```

```
e
s
t
i
m
a
d
o
s
```

Al analizar el ejemplo vemos que la variable *palabra* que contiene una cadena de caracteres, funciona como una secuencia, y la variable *letra* en cada iteración toma automáticamente el caracter subsiguiente.

1.2.7 Iteraciones sobre secuencias numéricas

Para iterar sobre secuencias numéricas combinamos el uso del *for* con la función *range()*. Veamos un ejemplo de una iteración sobre 3 valores:

```
for num in range(3):
    print(num)
```

```
0
1
2
```

Cuando utilizamos la función *range()* con un único argumento como dato, por ejemplo *tres*, nos genera una secuencia de tres valores, comenzando desde cero y avanzando de a un valor por vez, es decir, con paso uno.

Es posible cambiar este comportamiento indicando el valor inicial y final haciendo *range(inicio, fin)*, por ejemplo, se se desea iterar por valores numéricos entre 10 y 14:

```
for num in range(10,14):
    print(num)
```

```
10
11
12
13
```

Se debe notar que el valor final no es alcanzado en la iteración. También es posible indicarle el paso del incremento, como se deduce del ejemplo previo, al indicar solamente el valor inicial y final, se da por sentado que el incremento es de 1, cambiemos este comportamiento utilizando *range(inicio, fin, paso)*:

```
for num in range(10,19,2):
    print(num)
```

```
10
12
14
16
18
```

Otra posibilidad es recorrer una secuencia numérica en sentido sentido inverso, utilizando un incremento negativo y los valores de inicio y fin consistentes:

```
for num in range(19,10,-2):
    print(num)
```

```
19
17
15
13
11
```

Del resultado previo queda en evidencia que se mantiene la coherencia respecto a excluir el último valor de la secuencia y a incluir el inicial.

Veamos un ejemplo que resolvimos anteriormente utilizando el *while*, ahora usando *for*:

```
for vez in range(5):
    temperatura = int(input('Ingrese la temperatura en ºC:'))
    if (temperatura > 16):
        print('Vas caminando')
    else:
        print('Mucho frío, en vehículo')
```

Como vemos, nos despreocupamos de la inicialización de la variable *vez* y de controlar su incremento, ya que esto se realiza automáticamente en el *for*, por lo que para ciclos que conocemos de antemano la cantidad de iteraciones suele ser más simple y directo que el *while*.

La sentencia *for* en combinación con *range()* es una instrucción muy potente y flexible, más aún al ser combinadas con otro tipo de estructuras de datos como cadenas de caracteres y listas, que veremos en secciones posteriores.

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.

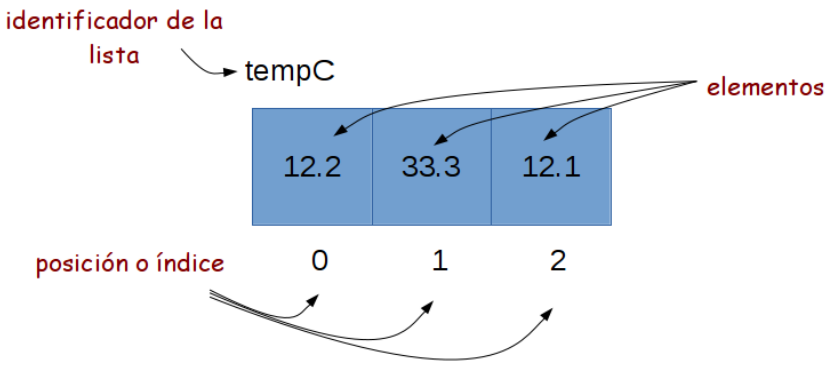
Si utilizamos variables, deberíamos leer los 120 valores para calcular el promedio y reingresar nuevamente las temperaturas mensuales para corroborar aquellas que superaron la media. Claramente el usuario de este programa no estará muy feliz de tener que reingresar la totalidad de los datos.

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
```

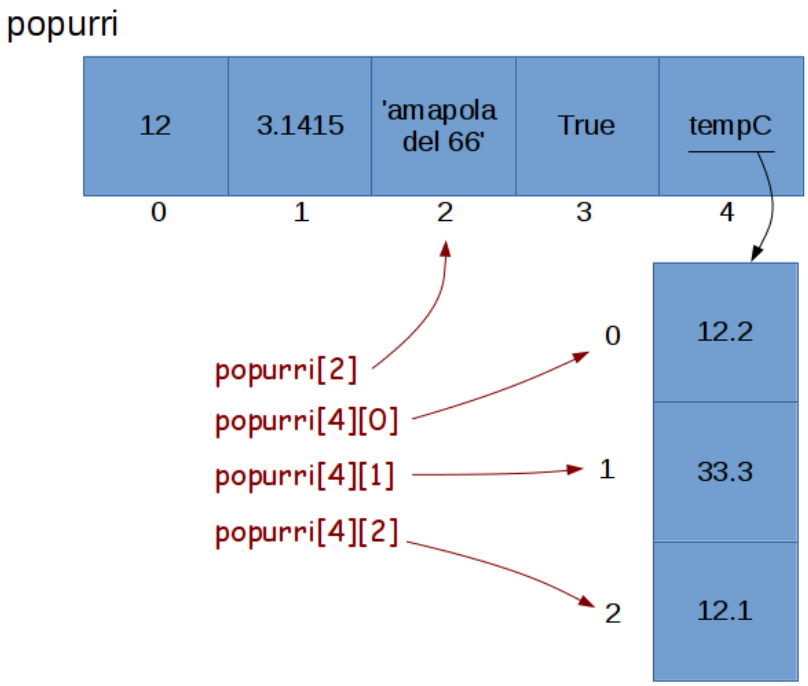
```
4to elemento:  True
5to elemento:  [12.2, 100, 12.1]
```

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.

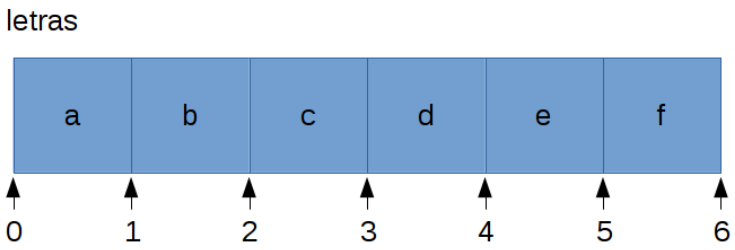
2.1.3 Rebanadas (slices)

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
```