Tecnicatura Universitaria en Software Libre

Introducción al Desarrollo de Software

Docente: Emiliano López
Tutor: Maximiliano Boscovich

Contenidos

1 Unidad 3: Estructuras de datos y control de flujo 3
1.1 Estructuras condicionales 3
1.1.1 Sentencia *if* 3
1.1.2 Sentencia *if..else* 4
1.1.3 Estructura de selección múltiple *if..elif..else* 5
1.1.4 Estructuras anidadas 6
1.2 Estructuras repetitivas 7
1.2.1 Sentencia *while* 7
1.2.1.1 Bucles condicionales 7
1.2.1.2 Bucles interactivos 9
1.2.1.3 Bucles centinelas 9
1.2.2 Sentencia *for* 11
1.2.3 Iteraciones sobre secuencias numéricas 11
1.3 Estructura de datos 13
1.3.1 Listas 13
1.3.1.1 Listas bidimensionales 16
1.3.1.2 Operaciones 17
1.3.1.3 Rebanadas (slices) 17
1.3.1.4 Métodos 18
1.3.2 Diccionarios 19
1.3.2.1 Operaciones 21
1.3.2.2 Métodos 21
1.3.3 Tuplas 22
1.3.4 Conversión entre listas y diccionarios 22
1.3.4.1 De diccionarios a listas 22
1.3.4.2 De listas a diccionarios 23
1.3.5 Cadenas de caracteres 23
1.3.5.1 Operaciones 24
1.3.5.2 Métodos 24

Introducción al Desarrollo de Software - Unidad 3

Este documento fue generado el 2015-09-15 15:08

1 Unidad 3: Estructuras de datos y control de flujo

Si un programa no fuera más que una lista de órdenes a ejecutar de forma secuencial, una por una, no tendría mucha utilidad. Es por ello que en la mayoría de los lenguajes de programación existen lo que se denominan estructuras de control. Estas estructuras permiten que, ante determinadas condiciones, un programa se comporte de diferentes maneras.

Supongamos que queremos hacer un programa que nos haga ciertas preguntas y en base a las respuestas determine si nos conviene ir al trabajo en bicicleta o en auto. Este programa prodría considerar inicialmente la temperatura ambiente, la hora y la distancia. Estos indicadores (variables), determinarán si el programa se debe comportar de una forma u de otra para de este modo recomendarnos una cosa (el uso de la bicicleta) u otra (el auto).

1.1 Estructuras condicionales

La primer estructura de control que veremos son los condicionales, los cuales nos permiten comprobar condiciones y hacer que se ejecute un fragmento de código u otro, dependiendo de esta condición. Aquí es donde cobra su importancia el tipo booleano que aprendimos en la sección anterior sobre los tipos básicos.

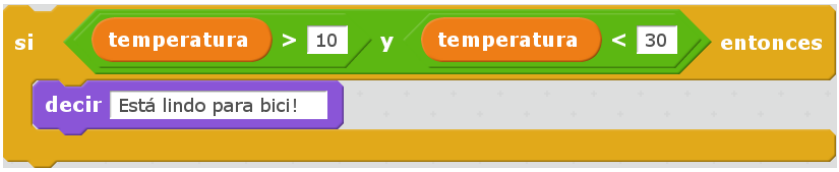
1.1.1 Sentencia if

La forma más simple de un estamento condicional es un ***if*** (del inglés si) seguido de la condición a evaluar, dos puntos (:) y en la siguiente línea e indentado(con sangría), el código a ejecutar en caso de que se cumpla dicha condición. Por ejemplo, si consideramos lo anterior, y hacemos que el programa por ahora solo considere la temperatura, podríamos hacer lo siguiente:

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

```
Deberías ser amable con el medio ambiente e ir en bicicleta
```

Esta sentencia se lee como: si (if) temperatura mayor a 10 y menor a 30, entonces ejecutar: print('Está lindo para bici!'). Estas sentencias solo se ejecutarán si se cumple la condición de que la variable temperatura contenga un valor que este entre 10 y 29, para el caso donde temperatura sea menor a 10 o mayor a 29, el programa no hará nada.



Una característica saliente para este tipo de comparaciones en Python es la de asemejar al lenguaje natural, por lo que podemos implementar la comparación previa haciendo:

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

¿Qué acciones se ejecutan al cumplirse la condición?

Una cuestión muy importante es indentar tal como se ha hecho en el ejemplo, es decir, aseguraros de dejar una sangría en las líneas debajo de los 2 puntos (:) que se deben ejecutar en caso que la condición de la pregunta se cumpla.

Todo lenguaje de programación tiene en su sintaxis un modo de identificar las acciones que forman parte de un bloque, en Python esto es a partir de la sangría.

1.1.2 Sentencia if..else

Nuestro interés inicial era que el programa nos dijera si podemos ir en auto o en bicicleta, y el ejemplo anterior solo nos dice algo cuando podemos ir en bici, pero no dice o hace nada cuando la condición no se cumple. Para estos casos existe un condicional llamado ***else***(del inglés si no), que se usa conjuntamente con if y que sirve para ejecutar ciertas instrucciones en caso de que la condición de la sentencia if no se cumpla. Por ejemplo:

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

```
Está lindo para ir en bici
```

Esto se lee como *si temperatura es mayor o igual a 10 y temperatura es menor que 30, entonces mostrar el mensaje 'Está lindo para ir en bici', sino mostrar el mensaje 'Te recomiendo ir en cole'*. Siempre se ejecutará una opción u otra, 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 (que en este caso es el valor de la variable temperatura).

En este caso también tenemos que prestar atención a la indentación utilizada. La sentencia *else* se escribe al mismo nivel que la sentencia *if*, y las sentencias que se deben ejecutar en caso de no se cumpla la condición if, deben ir indentadas también.

Una versión más completa del programa podría ser la siguiente:

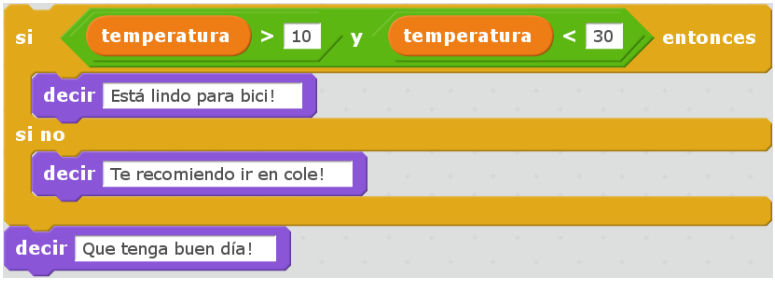
```
temperatura = int(input('Ingrese la temperatura en ºC:'))

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

print('Que tenga buen día!')
```

```
Ingrese la temperatura en ºC:12
Deberías ser amable con el medio ambiente e ir en bicicleta
Que tenga buen día!
```

En este caso consultamos por la temperatura, pidiendole al usuario que la ingrese por teclado (para esto utilizamos la función *input* que vimos en la Unidad 1). Luego mostramos en patalla lo que corresponda según el valor ingresado, y por último mostramos el mensaje 'Que tenga buen día!'. Es importante mencionar que la última sentencia siempre se ejecutará, la bifurcación se produce solamente entre las sentencias que estan dentro del if y el else, lo restante se seguirá ejecutando de manera secuencial.



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

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

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 de esta estructura es del siguiente modo:

```
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)
```

```
Ingrese la nota (0..100): 98
Calificación:  Distinguido
```

Como se observa, cada expresión condicional planteada es exluyente 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?

En 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, supongamos ahora que también queremos considerar la distancia que se debe recorrer. En este caso deberíamos preguntar por la distancia, pero también por la temperatura. Para que en los casos donde la temperatura sea agradable, la distancia no sea demasiado larga como para ir en bicicleta.

Para estos casos, se pueden utilizar estructuras anidadas, es decir, en el bloque de código que se ejecutará en caso de cumplirse o no una determina condición, podemos poner una nueva estructura de control, por ejemplo un nuevo *if*.

Reescribamos el código anterior para que considere esta nueva condición, y veamos como usar estructuras anidadas:

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

if (temperatura > 10) and (temperatura < 30):
    if (distancia <= 15):
        print('Está lindo para ir en bici')
    else:
        print('Está lindo, pero es lejos, le recomiendo ir en auto')
else:
    print('La temperatura no es agradable, le recomiendo ir en auto.')

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 este caso si se cumple la condición de que la variable temperatura contiene un valor entre 10 y 29, se pasa a considerar el valor de la variable distancia; si esta es menor o igual a 15, se muestra el mensaje *'Está lindo para ir en bici'*, en caso contrario, se muestra el mensaje *'Está lindo, pero es lejos, le recomiendo ir en auto'*. Por otro lado, si el valor de la variable temperatura no esta entre 10 y 29, se seguirá mostrando el mensaje *'La temperatura no es agradable, le recomiendo ir en auto'*. Lo mismo sucede con la última sentencia, la cual 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 ser ejecutada e inmediamente después 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, iniciandolo y esperando su finalización para repetir el proceso una y otra vez. Se hace evidente la ausencia de una estructura que permita repetir cuantas veces se requiera una determinada acción, 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 estructura 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 partes del ciclo *while*.

1.2.1.1 Bucles condicionales

Veamos un ejemplo donde se le pregunte el valor de temperatura a 5 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:

1. Leemos una temperatura que se ingresa por teclado
2. Escribimos en pantalla un mensaje según la temperatura
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 englobar 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 ésto, usamos una variable con un valor inicial conocido (1) 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
while vez <= 5:
    Pasos1y2
    vez = vez + 1
```

Ahora bien, cuando finaliza la ejecución de la instrucción `vez = vez + 1` la estructura iterativa evalúa nuevamente la expresión `vez <= 5` cuyo resultado puede ser cierto o no (`True` o `False`). Si el resultado es `True`, entonces el ciclo continuará con las acciones contenidas, re-evaluando la expresión en cada itereación 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 pequeño programa.

```
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 bucle, donde la cantidad de iteraciones depende de una condición es denominado como **bucles o lazos condicionales** y cuenta con dos características destacables:

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

El primer item evita obtener un mensaje de error, ya que no es posible evaluar una expresión con un valor que aún no ha sido definido, es decir, que no tiene asignado algún valor válido.

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.

1.2.1.2 Bucles interactivos

Otro tipo bucle para el que la estructura *while* se adapta fácilmente es aquellos donde la repetición depende de un valor que ingresa el usuario, es decir, para aquellos programas donde la condición de corte o repetición sea interactiva. Veamos un ejemplo en el que se calcula el promedio a partir del ingreso por parte del usuario de valores numéricos enteros.

Pensemos una posible estrategia para su solución: el programa le solicitará ingresar un nuevo valor numérico mientras que el usuario ingrese *si*, a su vez deberá ir sumando estos valores y contándolos. 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
    Pregutar al usuario si sigue ingresando números
Mostrar en pantalla el promedio
```

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

```
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.

1.2.1.3 Bucles centinelas

Otro tipo de bucles denominados 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, sin 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 amigable 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 ser cuidadoso en 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.

Para el ejemplo expuesto, la limitación esta dada para aquellos casos donde se ingresen valores negativos para ser incluídos en el cálculo del promedio. Sin embargo, Python provee herramientas que permiten salvar este inconveniente.

El problema consiste en:

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

```
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 igual al caracter vacío y no ingresará al ciclo *while*.

1.2.2 Sentencia for

La sentencia *for* provee otro modo de realizar bucles repetitivos en Python. Si bien la elección de un bucle u otro muchas veces dependerá del gusto o preferencia 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 de tipos de datos ya vistos, simplifando la definición, podemos definir una secuencia como todo tipo o 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.

1.2.3 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, para el ejemplo previo el número 3, nos genera una secuencia de 3 valores, comenzando desde 0 y avanzando de a un valor, es decir, con paso 1. Sin embargo, podemos 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, 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.

1.3 Estructura de datos

1.3.1 Listas

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 tipear nuevamente la totalidad de los datos.

Para este tipo de problemas y muchos otros más existe una estructuras más compleja y de gran utilidad denominada **lista**.

A diferencia de una variable que contiene un dato por vez, una lista puede almacenar varios en forma simultánea en diferentes posiciones, por lo que para referirnos a uno de ellos necesitamos especificarle el índice. 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:

12.2	33.3	12.1
0	1	2

Para **declarar e inicializar** una lista vacía y otra con esos tres valores haremos:

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

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 todo 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 i in tempC:
    print(i)
    media = media + i
media = media/3

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

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

Como se observa en el ciclo iterativo previo, las listas son perfectamente iterables en el `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 caracteres mientras que en una lista pueden ser de cualquier tipo y son llamados elementos o items.

Otro detalle es que una lista puede contener elementos de diferente 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, veamos el código:

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

```
12.2
100
12.1
```

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 items 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, fué utilizado como el valor para la función `range()` que generó una secuencia numérica (una lista!!!) que va desde 0 hasta `n-1`.

Veamos otro ejemplo de una lista de cadenas de caracteres. Tenemos algunos equipos de fútbol santafesino de primera división y queremos imprimir el fixture con todas las combinaciones de los partidos de ida, es decir, si el equipo A ya jugó con el B, no tendremos en cuenta que el equipo B juegue con el A.

Analicemos la estrategia: Por cada equipo de la lista debemos imprimir uno a uno los rivales subsiguientes, es decir, imprimimos el primer equipo con el segundo, luego con el tercero y finalmente con el cuarto. Luego, al pasar al segundo equipo de la lista, no debemos imprimir el primero, porque ya fué rival, sino que los restantes y así sucesivamente.

```
equipos = ["Colón", "A. Rafaela", "Central", "Newell"]
n = len(equipos)
for i in range(n):
    for j in range(i+1,n):
        print(equipos[i],"vs",equipos[j])
```

```
Colón vs A. Rafaela
Colón vs Central
Colón vs Newell
```

```
A. Rafaela vs Central
A. Rafaela vs Newell
Central vs Newell
```

1.3.1.1 Listas bidimensionales

Una lista unidimensional es aquella donde se utiliza un único índice para acceder a sus elementos, en el caso que utilicemos dos índices la lista es bidimensional y se la denomina matriz.

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 (1) de la tercer fila sería (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
()
```

1.3.1.2 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.

1.3.1.3 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, se pueden extraer conjuntos de elementos de la lista a partir de porciones o rebanadas (slices). Veamos unos ejemplos.

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

Podemos reemplazar varios elementos a la vez:

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

Además, puede eliminar elementos de una lista asignándoles la lista vacía:

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

Y se puede añadir elementos a la lista insertándolos en una porción vacía en la posición deseada:

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

1.3.1.4 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 `a[len(a):] = [x]`
- `list.extend(L)` Extiende la lista agregándole todos los ítems de la lista dada. Equivale a `a[len(a):] = 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 `a.insert(0, x)` inserta al principio de la lista, y `a.insert(len(a),x)` `equivale a a.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 a[:]`
- `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 `a[:]`

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]
```

1.3.2 Diccionarios

Hemos visto que las listas son útiles cuando se quiere agrupar valores en una estructura y acceder a cada uno de ellos a través del un valor numérico, un índice.

Otro tipo de estructura, que nos permite referirnos a un determinado valor a través de un nombre 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 esta 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', 'JPFeinman':'1523443', 'Dolina':'4584129',
          'Spasiuk':'65748', 'Fontanarrosa':'32456'}
```

Los *diccionarios* consisten en pares (llamados *items*) de *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 items son separados por comas, y toda la estructura es encerrada entre llaves. Un diccionario vacío, sin items, 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:

```
# Imprime claves
print("Claves")
print("======")
for nom in agenda:
    print(nom)
print()

print("Valores")
print("=======")
# Imprime valores
for tel in agenda.values():
    print(tel)
print()

print("Clave y valor")
print("=============")
# Imprime items: clave valor
for nom, tel in agenda.items():
    print(nom,tel)
```

```
Claves
======
JPFeinman
Spasiuk
Marado
Dolina
Fontanarrosa
```

```
Valores
=======
1523443
65748
1552123
4584129
32456

Clave y valor
=============
JPFeinman 1523443
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 items
- El valor de cada item es otro diccionario que a su vez contiene:
    - Tres items 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("Titulos")
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()
```

```
Titulos
=======
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
```

1.3.2.1 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 item 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.

1.3.2.2 Métodos

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

- *clear()* Elimina todos los items
- *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 item 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 items 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 items. Otras comparaciones ('<', '<=', '>=', '>') no son permitidas.

Para profundizar sobre diccionarios se recomienda la lectura del *Tutorial de Python* (pág. 32, *Diccionarios*).

1.3.3 Tuplas

Las tuplas son secuencias, al igual que las listas. La única diferencia es que no pueden ser modificadas, son inmutables (al igual que las cadenas de caracteres).

La sintaxis de las tuplas es simple, al separar varios valores con comas, automáticamente se crea una tupla.

```
t = 28, 21, 'hola!'
print(t[0])
print(t)

# desempaquetado de una tupla
x, y, z = t
```

```
28
(28, 21, 'hola!')
```

Para mayor detalle sobre esta estructura se recomienda leer el Tutorial de Python, *Tuplas y secuencias*, pag. 30.

1.3.4 Conversión entre listas y diccionarios

1.3.4.1 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 anteriomente:

```
print("Lista de items")
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 items
==============
[('Dolina', '4584129'), ('Fontanarrosa', '32456'), ('JPFeinman', '1523443'),
('Spasiuk', '65748'), ('Marado', '1552123')]

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

Lista de valores
===============
['4584129', '32456', '1523443', '65748', '1552123']
```

1.3.4.2 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)
```

```
{'JPFeinman': '1523443', 'Fontanarrosa': '32456', 'Dolina': '4584129',
 'Spasiuk': '65748', 'Marado': '1552123'}
```

1.3.5 Cadenas de caracteres

Una cadena es una secuencia de caracteres. Las hemos usado para mostrar mensajes, pero sus usos son mucho más amplios que sólo ése, a continuación las veremos mas en profundidad.

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 soporta el indexado y las rebanadas o porciones tal como las listas.

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:]`

1.3.5.1 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 entre si mediante los símbolos: >, >=, <, <=, ==, !=. Veamos un ejemplo:

```
palabra = input("Ingresá 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)
```

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

1.3.5.2 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
- `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
```