



INTRODUCCIÓN A LA PROGRAMACIÓN CON PYTHON



UNIDAD III

Estructuras de datos y control de flujo

MSc Ing Emiliano López

Introducción a la Programación con Python

Autor: Emiliano López - elopez@fich.unl.edu.ar

Fecha: 19/10/2016 01:20 - [última versión disponible]

Unidad III: Estructuras de datos y control de flujo

1 Control de flujo	4
1.1 Estructuras condicionales	4
1.1.1 Sentencia <i>if</i>	4
1.1.2 Sentencia <i>if..else</i>	5
1.1.3 Estructura de selección múltiple <i>if..elif..else</i>	6
1.1.4 Estructuras anidadas	8
1.2 Estructuras repetitivas	8
1.2.1 Sentencia <i>while</i>	9
1.2.2 Bucles condicionales	9
1.2.3 Bucles interactivos	11
1.2.4 Bucles centinelas	12
1.2.5 Bucles infinitos y break	13
1.2.6 Sentencia <i>for</i>	13
1.2.7 Iteraciones sobre secuencias numéricas	14
2 Estructuras de datos	17
2.1 Listas	17
2.1.1 Listas bidimensionales	20
2.1.2 Operaciones	22
2.1.3 Rebanadas (slices)	22
2.1.4 Métodos	23
2.2 Diccionarios	24
2.2.1 Operaciones	27
2.2.2 Métodos	27
2.3 Tuplas	28
2.4 Conversión entre listas y diccionarios	28

2.4.1	De diccionarios a listas	29
2.4.2	De listas a diccionarios	29
2.5	Cadenas de caracteres	31
2.5.1	Operaciones	31
2.5.2	Métodos	32

LICENCIA CC BY-SA 4.0



Introducción a la Programación con Python por Emiliano López se distribuye bajo una **Licencia Creative Commons Atribución-CompartirIgual 4.0 Internacional**.

A continuación una traducción de la licencia que podría diferir de la [original](#):

Usted es libre para:

- Compartir - copiar y redistribuir el material en cualquier medio o formato
- Adaptar - remezclar, transformar y crear a partir del material

Para cualquier propósito, incluso comercialmente

El licenciatario no puede revocar estas libertades en tanto usted siga los términos de la licencia

Bajo los siguientes términos:

- Atribución - Usted debe darle crédito a esta obra de manera adecuada (ver *), proporcionando un enlace a la licencia, e indicando si se han realizado cambios (ver **). Puede hacerlo en cualquier forma razonable, pero no de forma tal que sugiera que usted o su uso tienen el apoyo del licenciatario.
- Compartir Igual - Si usted mezcla, transforma o crea nuevo material a partir de esta obra, usted podrá distribuir su contribución siempre que utilice la misma licencia que la obra original.

* Si se suministran, usted debe dar el nombre del creador y de las partes atribuidas, un aviso de derechos de autor, una nota de licencia, un aviso legal, y un enlace al material. Las licencias CC anteriores a la versión 4.0 requieren que usted provea el título del material si se incluye, y pueden tener otras ligeras diferencias.

** En 4.0, debe indicar si ha modificado el material y mantener una indicación de las modificaciones anteriores

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
    ...
    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 ejecutarán, 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 °C: '))
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 °C: '))
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 °C: 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 °C: '))
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 °C:'))
    if (temperatura > 16):
        print('Vas caminando')
    else:
        print('Mucho frío, en vehículo')
    vez = vez + 1
```

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

```
Iniciar variable suma para sumar los números
Iniciar variable cant para contar los números
Iniciar 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 carácter 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 carácter 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, si 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 contrario, 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 °C: '))
    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.