Curso Básico de Python

3. Control de Flujo

Contenido

Esta unidad describe las estructuras de control usadas para definir qué sentencias se ejecutan, cuántas veces, en qué condiciones, cómo sigue el proceso después. Las sentencias están descritas en una versión simplificada; algunas de ellas tienen estructuras opcionales más complejas. Los ejemplos mencionados están disponibles en el sitio y pueden correrse como módulos.

Lecturas recomendadas:

1. Tutorial de Python, sección 3.2 Primeros pasos hacia la programación. En el capítulo 4 Más herramientas de control de flujo, secciones 4.1 a 4.5. La sección 4.6 trata la sentencia `match`, disponible recién a partir de la versión de Python 3.10, aún no incluida en muchas distribuciones ni en los intérpretes en línea, por lo que no la tratamos en esta versión del curso.

1 Sintaxis

Las sentencias de Python deben escribirse respetando reglas estrictas, único modo para lograr la interpretación correcta por parte de una máquina. El conjunto de reglas para escribir correctamente las instrucciones en un lenguaje de programación se denomina *sintaxis*. Python provee informacióin sobre la sintaxis de sus comandos a través del comando `help`. Por ejemplo, el comando `help("if")` nos muestra la siguiente salida:

```
The "if" statement is used for conditional execution:

   if_stmt ::= "if" assignment_expression ":" suite
               ("elif" assignment_expression ":" suite)*
               ["else" ":" suite]
```

Aquí nos dice que la sentencia `if` se usa para la ejecución condicional, es decir, la ejecución de un bloque de sentencias (*suite*) si se cumple una cierta condición, y solo en ese caso. La sintaxis está expresada como una fórmula: `if_stmt` es el nombre de esta definición, y el símbolo `::=` nos indica que esa definición viene a continuación. Las palabras reservadas y caracteres especiales van entre comillas; el resto de las palabras, como `suite` (bloque de sentencias) y `assignment_expression` (expresión de asignación) son construcciones que debemos proveer nosotros, siguiendo también las reglas del lenguaje. Lo encerrado entre paréntesis seguido de un asterisco `( )*` indica algo

opcional y que puede repetirse; el texto entre paréntesis rectos `[ ]` es opcional por una vez a lo más. La expresión de asignación del `if` y de los `elif` actúan como condiciones, deben resultar en verdadero o falso; el bloque siguiente solo se ejecuta si el resultado es verdadero. Estas convenciones forman parte del *metalenguaje*, el conjunto de símbolos y la forma en que se describe cómo deben escribirse las sentencias.
Observación: los bloques de sentencias deben escribirse indentados, dejando un número fijo de espacios al empezar el renglón (en español, *sangría*). Se recomienda usar 4 espacios para cada bloque indentado.

La ayuda en línea con el comando `help` está disponible para todas las sentencias. En esta unidad usaremos una notación simplificada para introducir la sintaxis.

2 Sentencia `if`

Sintaxis, del comando `help("if")` en notación simplificada:

```
if condición:
    bloque de sentencias
(elif condición:
    bloque de sentencias)*
[else:
    bloque de sentencias]
```

Cada bloque de sentencias se ejecuta solo si la condición bajo la cual se encuentra es verdadera. El siguiente ejemplo determina si un número es par:

```
num = input("Ingrese un número entero: ")    # ingresa como string
num = int(num)                               # convierte a entero
if num % 2 == 0:                             # si el resto es 0...
    print(num, "es par")                     # ...entonces es par
else:                                        # else es opcional; si no...
    print(num, "es impar")                   # ... es impar
```

El siguiente ejemplo muestra la sintaxis completa, con varias alternativas:

```
dia = input("Día de la semana: ")

if dia == "sábado":
    print(dia, ": medio horario")
elif dia == "domingo":
    print(dia, ": no laborable")
elif dia == "1 de mayo" or dia == "25 de agosto":
    print(dia, ": feriado no laborable")
elif dia in ["lunes", "martes", "miércoles", "jueves", "viernes"]:
    print(dia, ": laborable")
else:
    print("No es un día válido:", dia)
```

Módulos ejemplo : `ifelse_demo.py`, `if_par.py`, `if_diasemana.py`.

3 Sentencia `while`

Sintaxis, del comando `help("while")` :

```
while condición:
   bloque de sentencias
[else:
   bloque de sentencias]
```

El primer bloque de sentencias se ejecuta reiteradamente mientras la condición sea verdadera. Si la cláusula opcional `else` está presente, en cuanto la condición deja de ser verdadera se ejecuta el segundo bloque de sentencias, solo una vez, antes de terminar la sentencia.

Ejemplo sin `else`:

```
while cont < 5:
   print("Este mensaje se imprime 5 veces")
   cont += 1       # incrementa el contador
```

El mismo ejemplo con `else`:

```
while cont < 5:
   print("Este mensaje se imprime 5 veces")
   cont += 1       # incrementa el contador
else:
   print("Bucle terminado")
```

Módulo ejemplo: `while_mensaje.py`.

4 Sentencia `for`

Sintaxis, versión simplificada del comando `help("for")`:

```
for elemento in secuencia:
    bloque de sentencias
else:
    bloque de sentencias
```

El bloque de sentencias se ejecuta para cada uno de los elementos de la secuencia. Si la cláusula opcional `else` está presente, se ejecuta el segundo bloque de sentencias, una sola vez, antes de terminar.
Ejemplo sin `else`:

```
estaciones = ["primavera", "verano", "otoño", "invierno"]
for estacion in estaciones:
   print(estacion)
```

El mismo ejemplo con else:

```
estaciones = ["primavera", "verano", "otoño", "invierno"]
for estacion in estaciones:
   print(estacion)
else:
   print("estas son todas las estaciones")
```

Módulos ejemplo: `for_lista`, `for_estaciones.py`.

5 Sentencia `range`

Sintaxis, del comando `help("range")`:

```
range(fin)
range(inicio, fin, [paso])
```

Devuelve un objeto `range` que produce una secuencia de números enteros, desde *`inicio`* inclusive hasta *`fin`* (no incluído), incrementado en *`paso`*. Si se omite *`paso`*, su valor es 1.
Ejemplos:

```
for i in range(5):          # i recorre 0 1 2 3 4
   print(i, end=" ")
for i in range(5, 10):      # i recorre 5 6 7 8 9
   print(i, end=" ")
for i in range(5, 15, 2):   # i recvorre 5 7 9 11 13
   print(i, end=" ")
```

La sentencia `range` puede usarse para recorrer una lista:

```
estaciones = ["primavera", "verano", "otoño", "invierno"]
```

```
for i in range(len(estaciones)):  # i de 0 a largo de lista - 1
    print(estaciones[i])          # imprime posición i en la lista
```

Si bien lo anterior funciona, para recorrer una lista es más claro usar `for` *`item`* `in` *`lista`*`:`, como se mostró en la sección anterior sobre la sentencia `for`.

Módulo ejemplo: `range_secuencias.py`.

6 Sentencias `break` y `continue`

La sentencia `break` solo puede aparecer en un bucle o lazo repetitivo como `for` o `while`: termina inmediatamente la ejecución del bucle y continúa con la sentencia siguiente al `for` o `while`. Es útil para evitar continuar ejecutando el bucle cuando ya se cumplió alguna condición.

```
# break_frutas.py: pide un nombre de fruta y lo busca en una lista

frutas = ['uva', 'pera', 'melón', 'manzana', 'banana', 'naranja', 'palta', \
  'sandía', 'kiwi']

buscada = input("Fruta a buscar: ")
buscada = buscada.lower()  # convierte a minúscula, por si vino en mayúscula

hallado = False

for fr in frutas:
  if fr == buscada:
    hallado = True;
    break           # evita continuar recorriendo la lista inútilmente

if hallado:
  print ("'" + buscada + "'" + " está en la lista")
else:
  print ("'" + buscada + "'" + " no está en la lista")
```

Análogamente, la sentencia `continue` solo puede aparecer en un bucle o lazo repetitivo como `for` o `while`: en cuanto aparece, se pasa al siguiente item en el bucle, sin ejecutar el resto del bloque contenido dentro del `for` o `while`. Es útil para evitar procesamiento innecesario dentro del bucle cuando se ha cumplido una condición.

```
# continue_letras.py: cuenta la cantidad de letras 'p' o 'P' en un texto.

palabras = "Pedro Picapiedra padre picado pePino malpaso paPa paleta tapado"
cant = 0       # cantidad de p o P encontradas

for lt in palabras:
  if lt != 'p' and lt != 'P':
    continue   # vuelve al lazo repetitivo, en el siguiente item
  cant += 1    # incrementa la cantidad encontrada

print("Hay " + str(cant) + " letras 'p' o 'P' en el texto:")
print("   '" + palabras + "'")
```

Módulos ejemplo: `break_frutas.py`, `continue_letras.py`.

7 Sentencia `pass`

La sentencia `pass` no hace nada, solo se la usa cuando la sintaxis requiere una sentencia, pero no es necesario hacer nada en ese punto.

Módulo ejemplo: `for_while_demo.py`.

8 Sentencia `match`

Incorporada en Python 3.10; no la tratamos aquí. Muchas distribuciones vienen aún con la versión 3.8, donde esta sentencia aún no está incluida.

9 Ejercicios

1. Estudia y corre los programas ejemplo de este capítulo.

2. Estudia y corre el ejemplo `while_copia.py`. Agrega al ejemplo una cláusula `else`.

3. Arma una lista `ls_meses` con los nombres de los meses del año. Escribe código para:
   1. Recorrer la lista con `for`, usando una variable de caracteres llamada `mes`.
   2. Recorrer la lista con `for`, usando como índice un entero `nro_mes` y la función `len`.
   3. ¿Cómo puedes hacer para recorrer la lista usando `while`?
   4. Examina ahora el ejemplo `if_nromes.py`.

4. Escribe código para:
   1. Mostrar los números del 0 al 10.
   2. Mostrar los números del 0 al 10 de 2 en 2.
   3. Mostrar los números del 10 al 30.
   4. Mostrar los números del 10 al 30 crecientes de a 2.
   5. Mostrar los números del 30 al 10 decrecientes de a 2.

5. La función `random` del módulo `random` produce números aleatorios entre 0.0 y 1.0. El siguiente código genera una lista de números aleatorios entre 0 y 100:

```
from random import random    # generador aleatorio entre 0 y 1
ls_nros = []                 # lista vacía
for i in range(30):
    nro_decimal = random()               # nro aleatorio entre 0.0 y 1.0
    nro_entero = int(nro_decimal * 100)  # entero entre 0 y 99
    ls_nros = ls_nros + [nro_entero]     # agrega entero a la lista
print(ls_nros)
```

   1. Escribir un programa que pida un número entero entre 0 y 100, y busque en la lista ese número. En cuanto lo encuentre, salir del lazo. Si no lo encuentra, deberá escribir un mensaje apropiado.
   2. Lo mismo, pero mostrando todas las apariciones del número ingresado.

6. La lista anterior de números aleatorios puede ordenarse escribiendo `ls_nros.sort()`. Escribir un programa que pida un entero entre 0 y 100, y lo busque recorriendo la lista ordenada. Al encontrar el número pedido en la lista, salir del lazo inmediatamente (sentencia `break`). Al verificar que un número de la listan no es el buscado, continuar inmediatamente con el siguiente número en la lista (sentencia `continue`). Si el número biscado no está en la lista, imprimir un mensaje adecuado.

7. Estudia y corre el programa ejemplo `multiplos.py`. Pide por ejemplo buscar múltiplos de 13, en cantidad de 2, 10 y 20, y luego múltiplos de 130 (no habrá). Observa especialmente el uso de `break`, `continue` y `pass`, así como el `else` en el `for`.

8. Considera nuevamente el ejemplo `multiplos.py`. Busca un conjunto de múltiplos y cantidad de apariciones para probar todas las posibilidades del programa: que el múltiplo aparezca las veces pedidas, que aparezca pero no en la cantidad pedida, que no aparezca.

Diselar un conjunto de pruebas que cubra todas las posibilidades es un requerimiento imprescindible y no trivial para el desarrollo de software.