Curso de Python para Analisis de Datos

Clase 2

Indice:

Operaciones lógicas

Las operaciones lógicas son tan comunes como las comparaciones en cualquier programa, puesto que nos permiten crear comparaciones más complejas. Consideremos la siguiente variable:

In [ ]:
```
a = 5
```

¿Cómo puedo determinar si a es mayor a 1 y menor a 10, o, lo que es lo mismo, si a se encuentra entre 1 y 10? Necesito combinar dos operaciones de comparación; para ello, empleamos la palabra reservada (esto es, palabras que Python reserva para sí y que no pueden constituir el nombre de ninguna variable) `and` .

In [ ]:
```
a > 1 and a < 10
```

Esta operación se conoce como **conjunción** . El resultado de una conjunción es `True` cuando las dos condiciones que se propone conectar son verdaderas. En este caso, a > 1 and a < 10 es verdadero únicamente cuando a > 1 y a < 10 son ambas verdaderas. Si alguna de ellas es `False`, el resultado también lo es. Confirmemos lo que acabamos de decir:

In [ ]:
```
True and True
```

In [ ]:
```
True and False
```

In [ ]:
```
False and True
```

Por otro lado, la **disyunción** nos permite saber si alguna de dos comparaciones es verdadera. Por ejemplo, si queremos saber si a es igual a 1 o bien igual a 10:

In [ ]:
```
a == 1 or a == 10
```

Una disyunción es verdadera cuando **al menos una** de las dos comparaciones que conecta es `True`. En nuestro ejemplo, tanto a == 1 como a == 10 son comparaciones `False`, puesto que a es 5 . Comprobemos del mismo modo esta definición:

In [ ]:
```
True or True
```

In [ ]:
```
True or False
```

In [ ]:
```
False or False
```

La **negación** , que ejecutamos con la palabra reservada not , invierte el valor lógico de un `booleano`.

In [ ]:
```
not True
```

In [ ]:
```
not False
```

Condicionales

Dijimos que el código de un programa es ejecutado por Python de izquierda a derecha y de arriba hacia abajo. Los condicionales nos permiten diferir la ejecución de una porción de código (es decir, una o más líneas) según se cumpla una condición u otra; o, lo que es lo mismo, hacer que un bloque de código se ejecute únicamente cuando se cumpla una condición. Es decir, los condicionales nos permiten modificar el `flujo del programa`. Observemos el siguiente código de ejemplo y vayamos analizando parte por parte.

In [ ]:
```
a = 12
if a > 10:
    print("a es mayor que 10.")
```

La primera línea define una variable a con el valor 12 , y la segunda línea indica que el código a continuación solo debe ejecutarse cuando la condición a > 10 sea verdadera. En efecto, si cambiamos `a = 12` por `a = 7` , veremos que, al correr el programa, la tercera línea será ignorada.

El bloque de código que será ejecutado dependiendo del resultado de la condición debe tener aplicada una sangría de cuatro espacios (esto también se conoce como `indentación`, que es una transliteración del inglés indentation). La sangría es necesaria para poder diferenciar el bloque que se encuentra dentro del condicional del resto del código. Por ejemplo:

In [ ]:
```
a = 12
if a > 10:
    print("a es mayor que 10.")
print("Hola mundo")
```

En este caso, le estamos indicando a Python que únicamente la tercera línea debe ejecutarse si a > 10 . En cambio, la cuarta está por fuera del condicional por cuanto no tiene sangría, y por ende se ejecute siempre independientemente del resultado de la condición.

Los condicionales, entonces, están constituidos por:

- la palabra reservada **if** ,
- una **condición o expresión** cuyo resultado debe ser True o False seguida de **dos puntos(:)** y,
- una o más **líneas con sangría**.

En Jupyter la sangría se aplica automáticamente a la siguiente línea al presionar `Enter` luego de indicar los dos puntos al final de la condición. También podemos insertar manualmente sangría presionando la tecla `Tab` o cuatro veces la `barra espaciadora`. La cantidad de código que podemos incluir dentro de un condicional es ilimitada. Incluso puede contener otras definiciones de variables y otros condicionales:

In [ ]:
```
a = 12
if a > 10:
    print("a es mayor que 10.")
    b = 5
    if b == 5 or b == 3.14:
        print("b es 5 o 3.14.")
```

Ahora bien, consideremos esta variante:

In [ ]:
```
a = 12
if a > 10:
    print("a es mayor que 10.")
else:
    print("a es menor o igual que 10.")
```

La sintaxis es bastante clara: la tercera línea se ejecutará cuando a > 10 sea verdadero; y la quinta, cuando sea falso. Nótese que las palabras reservadas **if** y **else** deben estar alineadas, es decir, deben tener la misma sangría (si bien en este ejemplo no tienen sangría en absoluto). Por lo dicho es evidente que el código anterior es similar al siguiente:

In [ ]:
```
if a > 10:
    print("a es mayor que 10.")
if not a > 10:
    print("a es menor o igual que 10.")
```

O bien a este otro:

In [ ]:
```
a = 12
if a > 10:
    print("a es mayor que 10.")
if a <= 10:
    print("a es menor o igual que 10.")
```

Nos resta la palabra reservada **elif** , que nos permite agregar más de una condición en un mismo condicional:

In [ ]:
```
a = 5
if a == 1:
    print("a es 1.")
elif a == 2:
    print("a es 2.")
elif a > 2 and a < 10:
    print("a es mayor que 2 y menor que 10.")
else:
    print("a es mayor o igual que 10.")
```

Así como en un condicional del tipo **if / else** únicamente se ejecuta un bloque de código, del mismo modo ocurre aquí. Cuando tenemos un condicional con múltiples condiciones (en donde la primera se escribe con if y el resto con **elif**), todas ellas son evaluadas por Python (por “evaluar” entendemos determinar si son `True` o `False`) en el orden en el que están dispuestas en el código. De este modo, si a == 1 es `True`, se ejecuta print("a es 1.") y el condicional termina; es decir, el resto de las condiciones no son consideradas. Pero si a == 1 es `False`, entonces se considera la segunda condición, a == 2 , y así sucesivamente hasta llegar al else en caso que todas las condiciones anteriores sean falsas.

Además de los mencionados hasta ahora, Python nos ofrece otras estructuras de datos que son más flexibles y permiten agrupar varios valores. Además muchas de ellas son mutables, es decir, permiten que asignemos o alteremos valores una vez definidas.

Listas

Las `listas` nos permiten agrupar varios objetos (de cualquiera de los cuatro tipos de dato básicos que vimos) en una misma variable. Cada uno de estos objetos se conoce como `elemento` . Para crear una lista vamos a escoger un nombre e indicar, entre corchetes y separados por comas, los elementos. Por ejemplo, el siguiente código crea distintos tipos de listas:

In [ ]:
```
#Lista de numeros primos
numeros_primos = [2, 3, 5, 7, 11]

#Lista de nombres (strings)
alumnos = ["Jorge", "Matias", "Juan", "Martin"]

#Listas de distintos tipos de datos
datos = [3.14, True, -1, False, "Hola mundo"]
```

La disposición de los `elementos` al momento de crear una lista es importante. Las listas son **colecciones ordenadas** de objetos. A cada uno de los elementos se le asigna un **índice** , que no es otra cosa que la posición en la que se encuentra en la lista, empezando del 0. El índice nos permite luego traer un elemento en particular de una lista cualquiera.

Para acceder a un `elemento` de una `lista` a partir de su `índice`, vamos a indicarlo entre corchetes luego del nombre de la lista.

In [ ]:
```
print(numeros_primos[0])
print(numeros_primos[1])
print(numeros_primos[2])
print(numeros_primos[3])
print(numeros_primos[4])
```

Hasta ahora vimos solo como acceder utilizando indices positivos, pero también podemos acceder usando **indices negativos**. El último caracter tendra el indice -1, el anteultimo -2 y así sucesivamente.

In [ ]:
```
#Imprime el ultimo elemento de la lista numeros_primos
numeros_primos[-1]
```

In [ ]:
```
#Imprime la sublista que va desde el tercer elemento hasta el cuarto elemento inclusive de la lista numeros_primos
numeros_primos[2:5]
```

In [ ]:
```
#Imprime la sublista que contenga los primeros tres elementos de la lista numeros_primos
numeros_primos[:3]
```

In [ ]:
```
#Imprime la sublista que contenga los ultimos dos elementos de la lista numeros_primos
numeros_primos[-2:]
```

.append()

Ahora bien, podemos agregar un elemento al final de la lista (luego de haberla creado) usando la sintaxis `nombre_lista.append(elemento)` . Por ejemplo:

In [ ]:
```
numeros_primos.append(13)
print(numeros_primos)
print(numeros_primos[-1])
```

.insert()

Ahora bien, podemos agregar un elemento al final de la lista (luego de haberla creado) usando la sintaxis `nombre_lista.append(elemento)` . Por ejemplo:

In [ ]:
```
numeros_primos.insert(2, 3.14)
```

En este caso, el elemento 3.14 se inserta en la posición 2 , es decir, entre los números 3 y 5 .

In [ ]:
```
print(numeros_primos[2])
print(numeros_primos)
```

Nótese que todos los elementos subsiguientes al que acabamos de insertar sufren un desplazamiento hacia la derecha. De modo que el número 13, que antes estaba en la quinta posición, ahora está en la sexta.

La operación `insert` es bastante menos frecuente que `append` .

del lista[indice]

La sintaxis para quitar un elemento de una lista es `del nombre_lista[indice]` . Por ejemplo, para remover el elemento que acabamos de insertar:

In [ ]:
```
del numeros_primos[2]
```

Del mismo modo los elementos de una lista se reorganizan luego de haber eliminado uno de ellos, a menos que haya sido el último. En particular, todos aquellos que estaban a la derecha del elemento removido se desplazan una posición hacia la izquierda.

In [ ]:
```
numeros_primos

# Ahora la posicion [2] vuelve a ser el 5
```

len(lista)

Otra operación frecuente sobre listas es obtener el número de elementos que contiene. Lo conseguimos vía la instrucción `len(nombre_lista)`.

In [ ]:
```
len(numeros_primos)
```

Existe una relación entre el número o la cantidad de elemento de una lista y la posición del último elemento. Puesto que los índices comienzan del cero, es evidente que el último será `len(nombre_lista) - 1` .

In [ ]:
```
numeros_primos[len(numeros_primos)-1]
```

A partir de esto es claro que entre los corchetes podemos colocar un número o cualquier expresión que retorne un número. Por ejemplo, los siguientes métodos son todos similares para acceder al cuarto elemento (es decir, aquel cuyo índice es 3).

In [ ]:
```
numeros_primos[3] == numeros_primos[2+1]
```

In [ ]:
```
indice=3
numeros_primos[3] == numeros_primos[indice]
```

Por último, para crear una **lista vacía** usamos dos corchetes sin elementos.

Esto puede resultar útil cuando no conocemos los elementos de la lista al momento de crearla, pero los iremos añadiendo vía `append` o `insert` en el transcurso del programa.

In [ ]:
```
lista_vacia = []
```

Matrices

Una lista puede contener otras listas como elementos. La sintaxis para lograr esto no se diferencia en modo alguno de lo que ya hemos aprendido.

In [ ]:
```
mi_lista = [[3.14, "Hola mundo"], [True, False, -5]]
```

Este código crea una lista con dos elementos, que a su vez son listas:

In [ ]:
```
print(mi_lista[0])
print(mi_lista[1])
```

Para acceder a los elementos de alguna de esas dos listas, simplemente agregamos otro par de corchetes:

In [ ]:
```
print(mi_lista[0][0])
print(mi_lista[0][1])
print(mi_lista[1][0])
print(mi_lista[1][1])
```

Las matrices no son un tipo de colección particular en Python, sino que pueden simularse usando listas que contengan otras listas dentro de sí. Por ejemplo:

In [ ]:
```
m1 = [[3.3, 6.1, 4.0], [4.9, 5.7, 6.4]]
```

Ahora, según lo que acabamos de ver con el ejemplo de mi_lista , accedemos a los elementos indicando, entre corchetes, primero la **fila** y luego la **columna**.

In [ ]:
```
m1[0][2]
```

In [ ]:
```
m1[1][1]
```

Bucle While

Los bucles son otra herramienta que permiten modificar el flujo de un programa. Vimos que los condicionales pueden diferir la ejecución del código según se cumpla una condición u otra. Asimismo, los bucles repiten un bloque de código en tanto en cuanto se cumpla una condición. El bucle “while” en particular repite una porción de código siempre que una expresión sea verdadera.

Observemos la sintaxis con el siguiente ejemplo:

In [ ]:
```
a = 1
while a < 5:
    print("Hola mundo")
```

Ejecutando esto veremos que nuestro programa imprime “Hola mundo” **infinitamente**. Esto ocurre justamente porque el bloque de código del bucle, es decir, `print("Hola mundo")`, se ejecuta siempre y cuando la condición `a < 5` sea verdadera. Puesto que a es 1 y no cambia su valor en todo el programa, la condición es siempre verdadera y el bucle se ejecuta infinitamente.

Para que un bucle sea útil debe terminar en algún momento. Hay dos formas de conseguirlo: que la condición se vuelva falsa, o ejecutar una instrucción para forzar al bucle a que termine.

Comencemos por la primera. Debemos hacer que `a < 5` sea falso en algún momento para que el bucle termine. Podemos lograrlo vía el siguiente código:

In [ ]:
```
a = 1
while a < 5:
    print("Hola mundo")
    a = a + 1
```

Ejecutemos este programa y veremos que el resultado ahora son cuatro **“**`Hola mundo`**”** impresos en pantalla. La lógica de este comportamiento se comprende fácilmente si entendemos cómo avanza un bucle `while`:

1. Chequear que la condición sea verdadera.
2. Si es verdadera, ejecutar el bloque de código.
3. Si es falsa, terminar el bucle (y también estos pasos).
4. Volver al paso 1. Así, nuestro bucle se ejecuta hasta que a se convierte en 5 , y en consecuencia la condición `a < 5` (esto es, 5 < 5) se vuelve falsa.

El segundo método para finalizar un bucle es vía la palabra reservada **break** . Cuando Python se encuentra con esta instrucción dentro de un bucle, éste termina abruptamente, por más que la condición siga siendo verdadera.

Nuestro código anterior puede replicarse usando `break` del siguiente modo:

In [ ]:
```
a = 1
while True:
    if a < 5:
        print("Hola mundo")
        a = a + 1
    else:
        break
```

Poner `True` en lugar de una condición hace sencillamente que el bucle se ejecute infinitamente. Luego nos encargaremos de terminarlo manualmente vía `break` cuando la condición (a < 5) sea falsa.

Bucle For

El bucle `for`, como tal, también repite la ejecución de un bloque de código, pero no en base a una condición, como en el bucle `while`, sino tantas veces como `elementos` contenga una `lista`.

El cuerpo del bucle `for` se ejecuta tantas veces como elementos tenga el elemento iterable. Cualquier iterable: listas, diccionarios, strings pueden ser utilizados.

Veamos su sintaxis con el siguiente ejemplo:

In [ ]:
```
alumnos = ["Matias", "Jorge", "Maria", "Gabriela"]
for alumno in alumnos:
    print("Hola mundo" )
```

Por el momento prestemos únicamente atención a lo siguiente: si corremos este programa, el bloque de código dentro del `for` - `print("Hola mundo")`- se ejecuta cuatro veces. Si agregamos un `elemento` a la `lista` de alumnos , veremos que se ejecutará cinco veces. Si agregamos otro más, seis. Y así sucesivamente. Los bucles `for` se aplican siempre sobre listas.

In [ ]:
```
alumnos = ["Matias", "Jorge", "Maria", "Gabriela"]
for alumno in alumnos:
    print(alumno)
```

Lo que ocurre es que en cada ejecución del bloque de código (que por la cantidad de `elementos` que tiene nuestra `lista` serán cuatro) la variable alumno contendrá un `elemento` de la lista alumnos diferente, siguiendo el orden en el que fueron definidos. En efecto, el código anterior es funcionalmente igual al a este:

In [ ]:
```
alumnos = ["Matias", "Jorge", "Maria", "Gabriela"]
alumno = alumnos[0]
print(alumno)
alumno = alumnos[1]
print(alumno)
alumno = alumnos[2]
print(alumno)
alumno = alumnos[3]
print(alumno)
```

La diferencia es que el bucle `for` se encarga de realizar las asignaciones a la variable alumno de forma automática.

Comparacion de bucles

La mayoría de las tareas que requieren repitencia de un código (para las cuales los bucles son la herramienta ideal) pueden solucionarse usando cualquiera de los dos bucles que vimos. Sin embargo, en ocasiones es más pertinente alguno de ellos en particular.

Supongamos que queremos imprimir números del uno al diez. Usar un `while` es una opción:

In [ ]:
```
i = 1
while i <= 10:
    print(i)
    i = i + 1
```

La alternativa es un bucle `for`:

In [ ]:
```
for i in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    print(i)
```

Range

Aquí es claro cuál de los dos métodos es más conveniente: si ahora queremos imprimir números del uno al cien, el bucle `while` solo necesita modificar su condición por `i <= 100` . Pero en el caso segundo caso hay que agregar los noventa números restantes a la lista. Para solucionar este último inconveniente, Python incluye una instrucción llamada **range** que genera listas de números enteros en tiempo de ejecución.

El código anterior, usando esta nueva instrucción, se expresa de la siguiente forma:

In [ ]:
```
for i in range(1, 11):
    print(i)
```

La instrucción `range(inicio, fin)` genera una lista de números desde inicio hasta fin-1 (de ahí que `range(1, 11)` produzca números del uno al diez). Ahora es mucho más fácil cambiar este código para imprimir números del uno al cien, simplemente reemplazando `range(1, 11)` por `range(1, 101)`.

Otro argumento que puede utilizarse en `range(inicio, fin, saltos)`, donde la tercer opcion seran cada cuantos numeros haran los saltos del conteo. Ejemplo:

In [ ]:
```
# Range con salto cada 2 numeros
for i in range(1, 11, 2):
    print(i)
```

Hasta ahora vimos cómo crear una lista por extensión (es decir, escribiendo los elementos de la lista al momento de crearla) y vimos cómo crear una lista vacía para luego ir agregándole elementos. Existe otra manera más prolija y eficiente de crear listas que se llama por comprensión. Es muy parecido a crear una lista con un for, pero se hace todo en una sola línea.

In [ ]:
```
lista = [num for num in range(1000)]
print(lista)
```

También podemos agregar condiciones y expresiones a las listas por comprensión

In [ ]:
```
lista = [num**2 for num in range(-10, 10) if ((num%2 == 0) | (num%3 == 0))]
print(lista)
```

La sintaxis completa de las listas por comprensión es la siguiente:

[*expresión* **for** elemento **in** *iterable* **if** *condición*]

A parte de ser más prolijas, las listas por comprensión son más eficientes.

Ejercicios:

- Realizar un programa donde el usuario ingresando la edad pueda saber si puede votar, y si su voto es obligatorio o no

- De dos números, saber cual es el mayor.

- Crea una variable numérica y si esta entre 0 y 10, mostrar un mensaje indicándolo.

- Añadir al anterior ejercicio, que si esta entre 11 y 20, muestre otro mensaje diferente y si esta entre 21 y 30 otro mensaje

- Realiza un programa que lea dos numeros por teclado y permita elegir entre 3 opciones en un menu:
  - Mostrar una suma de los dos numeros
  - Mostrar una resta de los dos numeros (el primero menos el segundo)
  - Mostrar una multiplicacion de los dos numeros
  - En caso de no introducir una opcion valida, el programa informara de que no es correcta.

- Crear un programa que permita ingresar dos cadenas vía la consola y luego las compare, imprimiendo un mensaje en caso que sean iguales y otro en caso que sean diferentes.

- Crear un programa que solicite el nombre de un alumno a través de la consola, y luego chequee que no esté vacío. En caso de estarlo, debe imprimir un mensaje de error; caso contrario, imprimir un mensaje indicando que se ingresó correctamente.

- Escribe un programa en el que ingreses un numero y nos avise si es divisible por 5 y 7.

Crea un convertidor de grados Fahrenheit - Celcius.

`Formula : c/5 = f-32/9 [ where c = tempratura en celsius y f = temperatura en fahrenheit`

Resultado esperado :

- 60°C son 140 en Fahrenheit
- 45°F son 7 en Celsius

- Para mas ejercicios https://www.w3resource.com/python-exercises/python-conditional-statements-and-loop-exercises.php (https://www.w3resource.com/python-exercises/python-conditional-statements-and-loop-exercises.php)