DOCUMENTACIÓN CHECKPOINT 5

TOURÉ BARRERO HERRERA

ÍNDICE

1. ¿Qué es un condicional?

Una frase condicional es aquella que incluye y lleva consigo una condición o requisito, es decir, que expone una frase secundaria que será ejecutada sólo si la frase principal o condición se cumple.

En python los condicionales nos sirven para comparar dos valores y esa comparación nos da como resultado un valor booleano, es decir algo Verdadero o Falso.
Si una determinada condición se cumple se van a ejecutar un determinado número de acciones y si no se cumple, puede que se ejecuten otras acciones o que no se ejecute nada.

La sintaxis de los condicionales se realiza a través de la sentencia `if` que que equivale a *si…*, por tanto, para escribir nuestra condición primero redactamos `if` seguido de lo que debe ser Verdadero para que esta se lleve a cabo.

Por ejemplo, la frase *"Si 1 es mayor que 0…"* equivaldría a `if 1 > 0:` tras esto ponemos dos puntos y en la línea de debajo con la indentación adecuada establecemos lo que queremos que ocurra. Por ejemplo, para escribir la frase *"Si uno es mayo que cero, escribe (este número es mayor que cero)"*, en python escribiremos:

```
if 1 > 0:
    print("este número es mayor que cero")
```

A parte de establecer lo que queremos que pase si esta condición se cumple, también podemos definir lo que debe pasar si la condición no se cumpliera, esto se puede hacer de dos formas.

Por un lado podemos decir *"Si esto se cumple, haz esto y si no se cumple haz otra cosa"*. Es decir, si no se cumple mi condición inicial haz esto otro, y esto se llevaría a cabo a través de "else".

Por ejemplo:

```
if 1 > 0:
    print(“este número es mayor que 0”)
else:
    print(“este número no es mayor que 0”)
```

Pero también podemos añadir otro u otros condicionales más, esto sería a través de la construcción “else if”, que en python se abrevia en “elif”, de la siguiente forma:

```
num = 0

if num > 0:
    print(“este número es mayor que 0”)
elif num == 0
    print(“este número es 0”)
else:
    print(“este número no es mayor que 0”)
```

1.1. Operadores comparativos

Para establecer las distintas comparaciones en los condicionales tenemos los operadores comparativos, y estos son los siguientes:

- Igualdad: ==
- Desigualdad: !=
- Mayor que: >
- Menor que: <
- Mayor o igual que: >=
- Menor o igual que: <=

1.2. Operadores Ternarios

También elxiste el llamado Operador Ternario, el cual nos permite redactar frases con `if` y `else` pero no se usa de manera tan común como los condicionales porque el Código Zen de Python establece que *lo simple es mejor que lo complejo* y esta

forma de escritura es más dificil de leer que la que nos ofrecen los condicionales con el `if` statement.

De todas formas puede que haya momentos en los que necesitemos redactar nuestra frase con `if` en una sola línea de códico y los operadores ternarios nos facilitan esta tarea.

Para crear nuestro operador ternario vamos a establecer una variable, por ejemplo: `num = 3` y si quisieramos saber si nuestro número es positivo o negativo, podemos crear un operador ternario que nos diga justo esto.

Le vamos a pedir que si el numero escogido es positivo nos diga *“este número es positivo!”* y si no lo es nos diga *“este número no es positivo”*. Para ello redactaremos lo siguiente:

```
num = 3

num_es_posit = “este número es positivo!” if num > 0 else “este número no es positivo”
```

Como podemos comprobar, esta manera de escribir nuestra condición no es tan intuitiva como la siguiente (usando condicional):

```
num = 3
if > 0:
    print(“este número es positivo!”)
else:
    print(“este número no es positivo”)
```

1.3. Cadenas de texto

También podemos usar los condicionales en cadenas de texto con el operador **`in`** y su correspondiente negativo que es **`not in`** por ejemplo si queremos saber si una palabra aparece dentro de una frase concreta.

Si queremos saber si la palabra mesa está en la frase “la mujer se sentó a la mesa” podemos crear el siguiente

condicional: *si la palabra mesa está en la frase "la mujer se sentó a la mesa", escribe "palabra encontrada en la frase!" si no, escribe "palabra no encontrada en la frase"*. De la siguiente manera:

```
palabra = "mesa"

frase = "la mujer se sentó a la mesa"

if palabra in frase:
     print("palabra encontrada en la frase!" )
else:
     print("palabra no encontrada en la frase")
```

Sin embargo hay que tener en cuenta que estas búsquedas son sensitivas al uso de mayúsculas o minúsculas, es decir, si la palabra buscada fuera "*Mesa*" en la frase *"la mujer se sentó a la mesa"*, el sistema no reconocería *mesa = Mesa*, por tanto la condición no se cumpliría y nos devolvería *"palabra no encontrada en la frase"*. Para no tener este problema podemos convertir ambas cadenas de texto a minúsculas para hacer nuestro condicional, de la siguiente forma:

```
palabra = "Mesa"

frase = "La mujer se sentó a la mesa"

if palabra.lower() in frase.lower():
     print("palabra encontrada en la frase!" )
else:
     print("palabra no encontrada en la frase")
```

Y de esta forma tendríamos como resultado *"palabra encontrada en la frase!"*

1.4. Condicionales compuestos

También podemos crear condicionales compuestos es decir frases que lleven más de una condición. Esto podemos hacelo gracias a los operadores **and** y **or**. Por ejemplo, para decir: *si un número es mayor que -1 y menor que 1 escribe "este número es 0".* Para realizar este ejemplo necesitaríamos el operador **and** y sería de la siguiente forma:

```
num = 0

if num > -1 and num < 1:
    print("este número es 0")
```

Esto implica que ambas condiciones deben ser Verdaderas para que se lleve a cabo la acción, sim embargo tenemos otro operador, el **or** que podemos usar cuando solo necesitamos que una de las condiciones sea verdadera, es decir, cuando nos baste con que una de las condiciones se cumpla para que se realice la accion usaremos **or** de la siguiente manera:

```
num = 0

if num > 0 or num = 0:
    print("este número no es negativo")
else:
    print("este número es negativo")
```

También tenemos el **and not**, que sería lo opuesto a **and**, es decir, necesitamos que se cumpla la primera condición y que NO se cumpla la segunda, por ejemplo:

```
num = 11
if num > 0 and not num < 10:
  print("este número es mayor que 10")
```

2. ¿Cuáles son los diferentes tipos de bucles en Python? ¿Por qué son útiles?

Según la RAE un bucle es “Un proceso que se repite indefinidamente”. En programación, un bucle, por tanto será una secuencia de instrucciones de código que se ejecuta repetidas veces, hasta que la condición asignada a dicho bucle deja de cumplirse. Dichas instrucciones se aplican a colecciones de datos como listas, tuples, diccionarios, rangos y cadenas de texto en un proceso repetitivo llamado iteración.

En python existen dos tipos de bucle distintos: bucles for-in y while.

2.1. Bucles for-in

Un bucle for-in nos permite iterar un proceso hasta que haya sido aplicado a todos los elementos dentro de nuestra colección, es decir, si tenemos una lista con tres elementos, el proceso que hayamos implementado se repetirá 3 veces y tras esto el bucle acabará y el programa seguirá con normalidad, igualmente, si uestra coleccion tuviera 100 elementos el bucle se repetiriá 1’00 veces antes de continuar.

Este comportamiento es muy util para nosotros puesto que muchas veces desconoceremos la cantidad exacta de elementos sobre los que vamos a trabajar y este tipo de loop nos permite iterar sobre los elementos sin saberlo.

2. 1. 1. Sintaxis de los for-in loops

Para crear nuestro bucle for-in necesitamos una colección de elementos sobre la que iterar, para nuestro ejemplo pongamos que queremos imprimir cada nombre en una lista llamada *hombres*: `hombres = [“Pablo”, “Juan”, “Pepe”, “Marco”]`

Existe una convención que establece que si una colección tiene un nombre plural, para definir los elementos dentro de ella se emplee el singular de esa palabra. Es decir, si nuestra lista se llama *hombres*, la convención indica que para seleccionar los objetos dentro de ella utilizaremos la palabra *hombre*, en singular, aunque esto es simplemente una convención

y se puede llamar a los elementos de la manera que mejor convenga.

Para comunicarle a nuestro programa que *Por cada hombre en la lista hombres imprima el hombre* haremos un bucle for-in de la siguiente forma:

```
hombres =  [“Pablo”, “Juan”, “Pepe”, “Marco”]

for hombre in hombres:
     print(hombre)
```

La palabra *hombres* (en plural) es el nombre de la variable, es decir, de nuestra lista y por tanto esta palabra debe escribirse igual para que el programa entienda cuál es la coleccion que estamos utilizando. Sin embargo la palabra *hombre* (en singular) podremos sustituirla por cualquier otra cosa, por ejemplo x:

```
hombres =  [“Pablo”, “Juan”, “Pepe”, “Marco”]

for x in hombres:
     print(x)
```

Y obtendremos el mismo resultado.

Algo muy importante que debemos tener en cuenta es la indentación, ya que todo lo que vaya dentro de nuestro bucle debe ir indentado. Esta debe ser al menos 2 o 4 espacios en el lado izquierdo para que Python lo reconozca como un bloque de código.

2. 1. 2. Listas, tuplas y diccionarios

La forma de trabajar con los bucles for-in en listas es igual para tuples, sin embargo ya que los diccionarios están compuestos por una llave y un valor asociado a ella, necesitamos una manera de seleccionar ambos elementos. Para verlo de manera más clara vamos a hacer un ejemplo:

```
jugadores = {
  “Delantero”: “Ronaldo”,
  “Portero”: “Casillas”,
  “Centro”: “Marcelo”,
}
```

Ahora si queremos seleccionar ambos elementos escribiremos:

```
for posicion, jugador in jugadores.items():
     print(“posición”, posicion)
     print(“Nombre del jugador:”, jugador)
```

Como hemos hablado antes, los dos valores que ponemos en la función pueden ser la palabra que elijamos y python va a asignar la primera palabra al elemenento inicial, es decir a la llave y la segunda palabra al valor.

2. 1. 3. Cadenas de texto

Si queremos hacer un loop en cada una de las letras de una cadena de texto como si se tratase de una colección de caracteres haremos lo siguiente:

```
my_string = “Hola, que tal”

for letra in my_string:
     print(letra)
```

2. 1. 4. Rangos

Cuando queremos iterar sobre una colección un cierto número de veces pero no queremos que ese número sea determinado por la cantidad de elementos que hay en dicha colección podemos usar los rangos. Por ejemplo puedo decir:

```
for num in range(1, 10):
     print(num)
```

Aquí debemos tener en cuenta dos cosas; la primera es que la palabra “num” es el iterador, y como en los ejemplos anteriores, esta palabra puede ser la que deseemos. Por otro lado, el rango que hemos establecido(1, 10) tiene el 10 como tope, es decir, esta asignación hará que se impriman del 1 al 9 pero el 10 no se imprimirá puesto que es el límite de nuestro rango, igual que cuando trabajamos seleccionando y cortando elementos en una lista.

También podemos establecer los intervalos que queremos en nuestro rango, añadiendo un número más, por ejemplo, si queremos imprimir sólo los números pares:

```
for num in range (0, 11, 2)
```

2. 1. 5. Break y Continue

Cuando queremos parar o alterar el comportamiento de un loop basado en una condición podemos usar los operadores lógicos `break` y `continue`.

Por ejemplo, imaginemos que hemos creado una web y tenemos una lista de usuarios vetados, para esto necesitaremos crear un loop para encontrar un nombre en una lista y que cuando lo encuentre su comportamiento se altere:

```
my_list = [“Carlos”, “Miguel”, “Ibai”, “Adrián”]

for usuario in my_list:
    if usuario == “Miguel”:
        print(f“Lo sentimos{usuario}, usted está vetado”)
        continue
    else:
        print(f”{usuario} puede entrar”)
```

Sin embargo, si lo que queremos es que una vez encontrado el usuario vetado el loop se termine sin haber iterado sobre todos los elementos y que se continue con la siguiente linea de código podemos hacer lo siguiente:

```
my_list = [“Carlos”, “Miguel”, “Ibai”, “Adrián”]

for usuario in my_list:
     if usuario == “Miguel”:
          print(f“{usuario} ha sido encontrado en el
índice{my_list.index(usuario)}”)
          break
```

Por tanto, la diferencia entre `break` y `continue` es que con `continue`, una vez que se realiza la condición que hemos establecido, el loop se sigue iterando con normalidad, sin embargo, con `break`, una vez que se cumple nuestro requisito, el loop se termina y se continua en la siguiente línea de código.

2.2. Bucles while

El bucle while se diferencia del **for-in** en que este tiene un número indeterminado de iteraciones, es decir, mientras que los bucles **for-in** (contando con que no haya alteradores como `break`) solo iteran hasta legar al fin de la colección, sin embargo, los bucles while iteran indefinidamente hasta que nosotros le digamos que pare, y si no se lo decimos, seguirá iterndo infinitas veces y puede llegar a romper nuestro programa.

Se pueden usar para iterar sobre colecciones de elementos, rangos, etc. de manera idéntica al **for-in** loop y como norma general es este último el que usaremos en la mayoría de nuestros bucles, sim embargo hay ciertas ocasiones en la que los bucles while se adecúan más a nuestras necesidades.

3. ¿Qué es una lista por comprensión en Python?

En Python la comprensión de listas nos permite combinar o crear listas a partir de otras listas, tuples o colección.

Por ejemplo, si tenemos una lista con elementos que queremos añadir a otra lista, haremos lo siguiente:

```
mi_lista = [100, 0, 30, 50]
mi_nueva_lista = [16, 47, 98, 99, 10]

for num in mi_lista:
    mi_nueva_lista.append(num)


print(mi_nueva_lista)
```

Si en vez de añadir los elementos de nuestra primera lista tal cual, quisieramos alterarlos de alguna manera, en este caso dividirlos entre 2, haremos lo siguiente:

```
mi_lista = [100, 0, 30, 50]
mi_nueva_lista = [16, 47, 98, 99, 10]

for num in mi_lista:
    mi_nueva_lista.append(int(num/2))


print(mi_nueva_lista)
```

4. ¿Qué es un argumento en Python?

Cuando creamos una función, por ejemplo:

```
def mi_funcion (a, b):
    print(a+b)

mi_funcion(3, 5)
```

Los argumentos son la información que introducimos a nuestra función para que realice su trabajo, en este caso los argumentos serán “a” y “b”, o lo que es lo mismo, 5 y 3. Estos

se especifican entre paréntesis tras el nombre de la función y podemos añadir tantos elementos como deseemos a nuestra función, siempre y cuando estén separados por comas.

5. ¿Qué es una función Lambda en Python?

La expresiones Lambda, también conocidas como funciones anónimas, son funciones generalmente simples o pequeñas que sirven para realizar un trabajo de manera rápida y que pueden ser añadidas a funciones más complejas.

Se llaman así puesto que no es necesario darles un nombre formal. Su comportamiento es como el de cualquier función pero no hay que declararla con la palabra clave def. Su sintaxis es la siguiente:

variable = lambda argumentos(separados por comas): expresión

Por ejemplo:

```
nombre_completo = lambda primero, segundo:
f”{primero}{segundo}”
```

Y podemos añadir esta expresión Lambda a otra función de la siguiente manera:

```
nombre_completo = lambda primero, segundo:
f”{primero}{segundo}”


def saludo(nombre):
    print(f”Hola{nombre}!”)

saludo(nombre_completo(“María”, “García”))
```

Las funciones Lambda pueden tener cualquier número de argumentos, pero solo una expresión.

6. ¿Qué es un paquete pip?

Pip es el sistema de gestión de paquetes utilizado para instalar y administrar paquetes de software escritos en Python.

Los paquetes pip son archivos de software agrupados en módulos y bibliotecas que pueden ser reutilizados en diferentes proyectos y proporcionan funcionalidades que van desde operaciones matemáticas hasta acceso a bases de datos y análisis de datos, entre otros.

La ventaja de usar un gestor de paquetes como pip es que simplifica el proceso de instalación y actualización de paquetes, así como la gestión de dependencias entre ellos.
Además, pip permite a los desarrolladores compartir sus propios paquetes con la comunidad y descargar paquetes creados por otros desarrolladores para usarlos en sus proyectos.

Se puede trabajar con los paquetes pip de la siguiente manera:

Instalar un paquete: pip install nombre_del_paquete
Actualizar un paquete: pip install –upgrade nombre_del_paquete
Desinstalar un paquete: pip uninstall nombre_del_paquete
Listar paquetes instalados: pip list

La ventaja de usar los paquetes pip es que puedes avanzar mucho más rápido al elaborar tus propios proyectos puesto que estás beneficiándote de todo el trabajo realizado ya por otros programadores y puedes centrarte en lo que realmente hace tu proyecto especial y valioso.