Checkpoint 5

Ainhoa Galán

Preguntas


1
Condicionales
2
Bucles
3
Listas por comprensión
4
Argumentos
5
Funciones Lambda
6
Paquete PIP

1. Condicionales en Python

Los condicionales son instrucciones que se le dan a un programa para que **ejecute un bloque de código u otro si se cumplen determinados requisitos**. De este modo, puede adaptarse a distintas situaciones e incluso responder ante usos o consultas inusuales.

El software entiende cada condición como **verdadera o falsa**, y actúa según dicha evaluación.

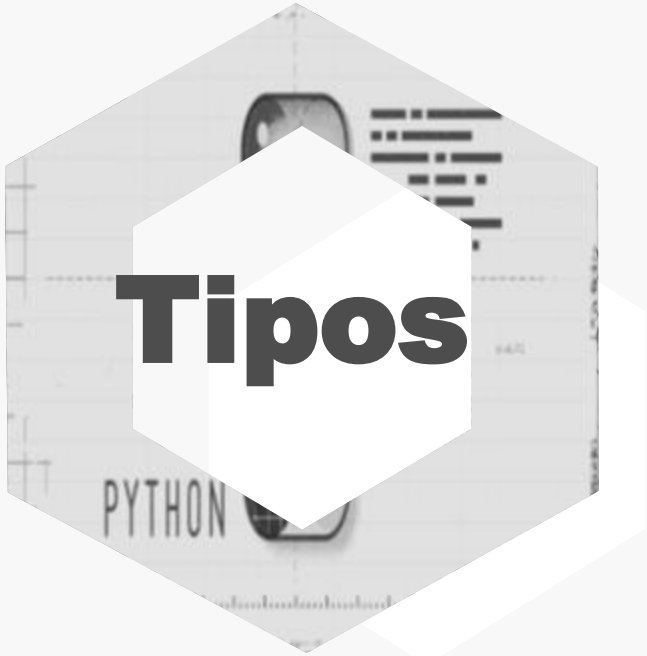
El software entiende cada condición como **verdadera o falsa**, y actúa según dicha evaluación. Para explicar las circunstancias que se valoran, se hace uso de los operadores de Python, especialmente de los **lógicos** —cuando hay que comprobar más de un requisito a la vez—, y los de **comparación** —cuando la condición es única.

operadores lógicos

‘and’ ‘or’ ‘not’

operadores de comparación

‘==’ ‘!=’ ‘<’
‘>’ ‘<=’ ‘>=’



Tipos

Los principales condicionales en Python son **if, else y elif**.
"Traducidos" al lenguaje humano, serían:

Condicional	*Significado*	*Ejemplo*
if	"Si"	edad = 18 if edad < 18: print("No puedes beber alcohol".)

else	"Si no"	edad = 18
		if edad < 18:
		  print("No puedes beber alcohol".)
		else:
		  print("Puedes beber alcohol".)

Condicional	*Significado*	*Ejemplo*

elif (else + if)	"Si no, si"	edad = 18
		if edad > 18:
		  print("Puedes beber alcohol".)
		elif edad == 18:
		  print("Como acabas de cumplir la mayoría de edad, puedes beber alcohol".)
		else:
		  print("No puedes beber alcohol".)

Según la tabla anterior, cada uno de los condicionales en Python **marca una ruta a seguir:**

- **if: ejecuta el código** al que va asociado **si la condición es verd**adera. En el ejemplo, sería mostrar el mensaje "No puedes beber alcohol" si el/la usuario/a tiene menos de 18 años.
- **else: va después de un if**. Aplica el código al que está conectado si la condición que el if marca es falsa. Es decir, si A no se cumple, else B.
- **elif:** si hay **varias condiciones que comprobar** antes de "decidir", se utiliza **if** en la primera y **elif** en las siguientes. En la tabla, expresa que, si el/la usuario/a no es mayor de 18 años, sino que tiene justo los 18, puede beber alcohol legalmente.

Uso del if

Un ejemplo sería si tenemos dos valores **a** y **b** que queremos dividir. Antes de entrar en el bloque de código que divide a/b, sería importante verificar que **b** es **distinto de cero**, ya que la división por cero no está definida. **Es aquí donde entran los condicionales if.**

```
a = 4
b = 2
if b != 0:
    print(a/b)
```

En este ejemplo podemos ver como se puede usar un if en Python. Con el operador **!=** se comprueba que el número b sea distinto de cero, y **si lo es, se ejecuta el código que está i**dentado. Por lo tanto **un if tiene dos partes**:

- **La condición** que se tiene que cumplir para que el bloque de código se ejecute, en nuestro caso b!=0.
- **El bloque de código** que se ejecutará si se cumple la condición anterior.

Es muy importante tener en cuenta que la sentencia if debe ir terminada por **:** y el bloque de código a ejecutar **debe estar identado**

Se puede también combinar varias condiciones entre el if y los :.
Por ejemplo, se puede requerir que un número sea mayor que 5 y además menor que 15. Tenemos en realidad tres operadores usados conjuntamente, que serán evaluados por separado hasta devolver el resultado final, que será True si la condición se cumple o False de lo contrario.
Es muy importante tener en cuenta que a diferencia de en otros lenguajes, en Python no puede haber un bloque if vacío.

```
a = 10
if a > 5 and a < 15:
    print("Mayor que 5 y menos que 15")
```

El siguiente código daría un SyntaxError.
if a > 5:
(para evitarlo habría que escribir después, "pass")

```
if a > 5:
    pass
```

Uso de else y elif

Es posible que **no solo** queramos hacer algo **si una determinada condición se cumple**, sino que **además** queramos **hacer algo de lo contrario**. Es aquí donde entra la cláusula **else**.

La parte **del if** se comporta como se ha visto antes, con la diferencia que **si esa condición no se cumple, se ejecutará el código** presente **dentro del else**. Ambos bloques de código son excluyentes, se entra o en uno o en otro, pero nunca se ejecutarán los dos.

```
x = 5
if x == 5:
    print("Es 5")
else:
    print("No es 5")
```

En muchos casos, **podemos tener varias condiciones diferentes y para cada una queremos un código distinto**. Es aquí donde entra en juego el **elif**

```
x = 5
if x == 5:
    print("Es 5")
elif x == 6:
    print("Es 6")
elif x == 7:
    print("Es 7")
```

Con la cláusula elif podemos ejecutar tantos bloques de código distintos como queramos según la condición.
Sería algo así como decir:
"si es igual a 5 haz esto, si es igual a 6 haz lo otro, si es igual a 7 haz lo otro".

Se puede usar también de manera conjunta todo, **el if con el elif y un else al final**. Es muy importante notar que **if y else solamente** puede haber **uno,** mientras que **elif** puede haber **varios**.

2.
Tipos de bucles

Al igual que los condicionales, los bucles son **estructuras de control** que sirven para modificar el flujo de un programa. En concreto los bucles lo que nos permiten es ejecutar un mismo código, de manera repetida, mientras se cumpla una condición. Existen dos tipos: el bucle **for** y el bucle **while**.

Bucle for

El **for** es un tipo de bucle, parecido al while pero con ciertas diferencias. **La principal es que el número de iteraciones de un for esta definido de antemano,** mientras que **en un while no**.

La **diferencia principal** con respecto al while es **en la condición**. Mientras que en el while la condición se evalúa en cada iteración para decidir si volver a ejecutar o no el código, en **el for no existe tal condición**, sino un iterable que define las veces que se ejecutará el código.

En el siguiente ejemplo vemos **un bucle for que se ejecuta 5 veces,** y donde la **i incrementa su valor "automáticamente" en 1** en cada iteración.

```
for i in range(0, 5):
    print(i)

# Salida:
# 0
# 1
# 2
# 3
# 4
```

En Python **se puede iterar prácticamente todo,** como por ejemplo una cadena. En el siguiente ejemplo vemos como la **i** va tomando los valores de cada letra

```
for i in "Python":
    print(i)

# Salida:
# P
# y
# t
# h
# o
# n
```

Bucle while

El bucle **while** nos permite ejecutar una sección de código repetidas veces, de ahí su nombre. El código se ejecutará mientras una condición determinada se cumpla. Cuando se deje de cumplir, se saldrá del bucle y se continuará la ejecución normal.
Llamaremos **iteración** a **una ejecución completa del bloque de código**.

```
x = 5
while x > 0:
    x -=1
    print(x)

# Salida: 4,3,2,1,0
```

En este ejemplo tenemos un **caso sencillo de while**. Tenemos una condición **x>0** y un bloque de código a ejecutar mientras dure esa condición **x-=1** y **print(x)**. Por lo tanto, mientras que x sea mayor que 0, se ejecutará el código. Una vez se llega al final, se vuelve a empezar y si la condición se cumple, se ejecuta otra vez. En este caso se entra al bloque de código 5 veces, hasta que en la sexta, x vale cero y por lo tanto la condición ya no se cumple.

Por lo tanto, el bucle while tiene dos partes:

- La **condición** que se tiene que cumplir para que se ejecute el código.
- El **bloque de código** que se ejecutará mientras la condición se cumpla.

¡Cuidado! Un mal uso del while puede dar lugar a bucles infinitos y problemas. En algún caso tal vez nos interese tener un bucle infinito, pero hay que tener cuidado.

3.
Listas por compresión

Una de las principales ventajas de Python es que una misma funcionalidad puede ser escrita de maneras muy diferentes, ya que su sintaxis es muy rica en lo que se conoce como expresiones idiomáticas. Las **list comprehension o comprensión de listas** son una de ellas.
Las *list comprehension* **nos permiten crear listas de elementos en una sola línea de código**. Por ejemplo, podemos crear una lista con los cuadrados de los primeros 5 números de la siguiente forma:

```
cuadrados = [i**2 for i in range(5)]
#[0, 1, 4, 9, 16]
```

De no existir, podríamos hacer lo mismo de la siguiente forma, pero necesitamos alguna que otra línea más de código. *El resultado es el mismo, pero resulta menos claro.*

```
cuadrados = []
for i in range(5):
    cuadrados.append(i**2)
#[0, 1, 4, 9, 16]
```

Sintaxis de las listas por compresión:

```
# lista = [expresión for elemento in iterable]
```

por un lado tenemos el for elemento in iterable, que itera un determinado iterable y "almacena" cada uno de los elementos en elemento y

por otro lado, tenemos la expresión, que es lo que será añadido a la lista en cada iteración.

4. Argumentos en Python

En Python, un argumento es la información que se pasa a una función cuando se llama a esta.

Ejemplo: En **def sumar(a, b):,** a y b son argumentos.

Los argumentos **se utilizan para proporcionar datos a las funciones para que puedan ejecutar las instrucciones.**

En cuanto a su sintaxis, los argumentos se escriben dentro de los paréntesis en la definición de la función y para utilizarlos, al llamar a la función se proporcionan los valores correspondientes a los argumentos que solicita esa función.

sumar(3,4)

Argumentos de entrada

Partimos de una fc. sin argumentos de entrada y la llamamos. Vemos que nos devuelve: 'Hola'

```
def di_hola():
    print("Hola")

di_hola() # Hola
```

Ahora le pasamos argumentos de entrada y valores

```
def di_hola(nombre):
    print("Hola", nombre)
di_hola("Juan")
# Hola Juan
```

Python permite pasar argumentos también de otras formas

Argumentos por posición

Los **argumentos por posición o posicional**es son la forma más básica e intuitiva de pasar parámetros. Si tenemos una función resta() que acepta dos parámetros, se puede llamar como se muestra a continuación.

```
def resta(a, b):
    return a-b
resta(5, 3) # 2
```

Argumentos por nombre

Otra forma de llamar a una función, es usando **el nombre del argumento con = y su valor.** El siguiente código hace lo mismo que el código anterior, con la diferencia de que los argumentos no son posicionales.

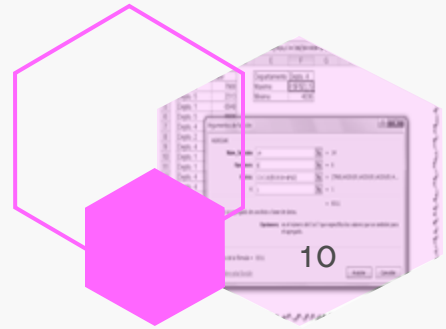
```
resta(a=3, b=5) # -2
```

Argumentos por defecto

A veces podemos tener una función con algún parámetro opcional, que pueda ser usado o no, dependiendo de las circunstancias. Para ello, lo que podemos hacer es asignar un valor por defecto.

En el siguiente caso *c valdría cero salvo que se indique lo contrario*.

```
def suma(a, b, c=0):
    return a+b+c
suma(5,5,3) # 13
```



5. Funciones Lambda

Las **funciones lambda o anónimas** son un tipo de funciones en Python que típicamente se definen en una línea y cuyo código a ejecutar suele ser pequeño. En la documentación oficial de Python dicen que:

"Python lambdas are only a shorthand notation if you're too lazy to define a function."

Lo que significa que: "*las funciones lambda son simplemente una versión acortada, que puedes usar si te da pereza escribir una función".*

Lo que sería una función que suma dos números como la siguiente.

```
def suma(a, b):
    return a+b
```

Se podría expresar en forma de una función lambda de la siguiente manera.

```
lambda a, b : a + b
```

La primera diferencia es que una función lambda no tiene un nombre, y por lo tanto salvo que sea asignada a una variable, es totalmente inútil. Para ello debemos.

```
suma = lambda a, b: a + b
```

Una vez tenemos la función, es posible llamarla como si de una función normal se tratase.

```
suma(2, 4)
```

Si es una función que solo queremos usar una vez, tal vez no tenga sentido almacenarla en una variable. Es posible declarar la función y llamarla en la misma línea.

```
(lambda a, b: a + b)(2, 4)
```

6. Paquetes PIP

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

- Los **paquetes** de software son **conjuntos de módulos y bibliotecas que pueden ser reutilizados en diferentes proyectos**. Los paquetes pueden **proporcionar 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.

Características clave de pip

1. **pip** facilita la instalación de paquetes de software. Puedes buscar paquetes en el Índice de Paquetes de Python (PyPI) y luego instalarlos con el comando: (pip install nombre_paquete)
2. Con **pip**, puedes actualizar fácilmente los paquetes
3. **pip** automáticamente instala las dependencias que un paquete necesita para funcionar.
4. **pip** también permite desinstalar paquetes que ya no necesitas. (pip uninstall nombre_paquete)
5. Con **pip** puedes obtener una lista de todos los paquetes que tienes instalados en tu sistema. (pip list)
6. **Y** además **pip** te permite instalar paquetes desde diferentes fuentes, como repositorios Git o archivos locales.

“Gracias"