

TEORÍA CHECKPOINT 5

Nélida Rodríguez

¿Qué es un condicional?

Es un concepto fundacional en torno a lo que hace que un programa se vuelva dinámico. Son procesos de sí/no. Este tipo de sentencias permite al programa tomar decisiones basadas en los valores de variables o en el resultado de comparaciones. Es fundamental aprender a usarlos para cualquier programador de Python.

Se escribe poniendo primero la condición `if` (sí en inglés) y `else` (otro en inglés) para el comparador. En caso de haber más de un comparador, en el medio se introduce el término `elif`, acrónimo de `else if`. Se pueden añadir cuantos `elif` se necesiten para el programa.

Sintaxis:

Condicional if-else

```
if condición:  
    expresión
```

```
else  
    expresión
```

Condicional if-elif-else

```
if condición:  
    expresión
```

```
elif condición:  
    expresión
```

```
else  
    expresión
```

Los condicionales también pueden estar anidados, añadiendo unos condicionales dentro de otros. Ejemplo de un condicional anidado:

```
Numero = 4 (número aleatorio)  
if numero == 2:  
    print("El número es el 2")  
  
else:  
    if numero == "6":  
        print("El número es el 6")  
  
    else:  
        if letter == "4":  
            print("El número es el 4")  
  
        else:  
            print("El número no es ni 2, ni 6, ni 4")
```

Operadores ternarios en los condicionales

El operador ternario o expresión condicional en Python nos permite acceder a una de dos opciones a partir de una condición.

Sintaxis:

```
opcion1 if condición else opcion2
```

Se evalúa la condición y se toma opcion1 si es cierta (True) o se toma opcion2 si es falsa (False).


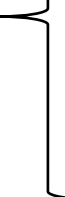
Emplear el operador ternario reduce las líneas de código, lo cual es importante en programación. Aun así, hay que tener cuidado a la hora de usarlo, ya que puede añadir complejidad innecesaria al código. Se debe uno cerciorar de que es la mejor opción de las que se tiene. Como buena práctica, se toma la mitad de la pantalla con la ventana donde se esté escribiendo el código. Si el ternario entra así en una línea, se puede usar. Si no, es mejor usar la forma tradicional.

Operadores de comparación

Los elementos de comparación se usan para diferentes cosas en los condicionales y devuelven un valor booleano. Se usan para:

- saber si un objeto es igual a otro o no.
- Para el valor de un rango, por ejemplo, si un elemento es menor o igual a otro.
- Resultados de verdadero – falso.

Los operadores de comparación son:

Funcionan con la mayor parte de objetos		==	Igualdad
		!=	Desigualdad
		<>	Desigualdad (obsoleto)
Funcionan con números		>	Mayor que
		>=	Mayor o igual a
		<	Menor que
		<=	Menor o igual a

Condicionales compuestos

Muchas veces se pueden hacer condicionales anidados para añadir múltiples condiciones en un programa de Python, pero son de difícil comprensión y/o añaden más líneas al código. Por lo que se utilizan condicionales compuestos. Pueden ser de dos tipos, como se observa en la sintaxis a continuación:

Sintaxis:Condicional if-and-else

```
if condicion1 and condicion2:
    expresión
```

```
else
    expresión
```

Condicional if-or-else

```
if condicion1 or condicion2:
    expresión
```

```
else
    expresión
```

Condicional if-and/or-and-else

```
if (condicion1 or condicion2) and condicion3:
    expresión
```

```
else
    expresión
```

Ejemplo de cómo funciona un condicional compuesto:

```
usuario = 'juana'
email = 'juana@mimail.com'
contraseña = 'lacasa'
```

```
if usuario == 'juana' or contraseña == 'lacasa':
    print('Puedes pasar')
```

```
else:
    print('No puedes pasar')
```

```
if usuario == 'juana' or contraseña == 'lacasa':
    print('Puedes pasar')
```

```
else:
    print('No puedes pasar')
```

```
if (usuario == 'juana' or email == 'juana@mimail.com') and
    contraseña == 'lacasa':
```

```
    print('Puedes pasar')
```

```
else:
    print('No puedes pasar')
```

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

En Python hay dos tipos de bucles: los `for/in`, que se usan el 95% de las veces, y los bucles `while`, útiles en casos específicos. La gran diferencia entre ellos es que con los `for/in` tienes un claro inicio y fin de bucle. Realiza la comprobación de la/s declaración/es tantas veces como elementos haya en la colección, lista, etc que esté evaluando. En los bucles `while`, sin embargo, la iteración no para cuando llega al final, sigue hasta que se le dice explícitamente que pare.

Bucles `for/in`

El bucle `for/in` de Python es más fácil de usar que los bucles en otros lenguajes de programación, de ahí su gran utilidad. En ellos, se itera directamente sobre los elementos de una colección. Dependiendo si es una lista, una tupla o un diccionario, la sintaxis varía.

- Bucles `for/in` con listas y tuplas

En estos bucles tenemos un elemento sobre el que se va a iterar en un diccionario.

Sintaxis: `for` variable `in` lista:
Expresión

La palabra elemento es aleatoria, simplemente hacen referencia a los elementos dentro de la colección en cuestión.

A continuación, un ejemplo de su uso:

```
casas = ['Caserío', 'Rascacielos', 'Masía']
```

```
for casa in casas:
    print(casa)
```

- Bucles `for/in` con diccionarios

Son similares a las de arriba, solo que en este caso tenemos un diccionario con llaves-valores. En vez de iterar sobre un elemento, se itera sobre dos, la llave y el valor.

Sintaxis: `for` llave, valor `in` diccionario.items():
expresión

Las palabras “llave” y “valor” son aleatorias, simplemente hacen referencia a las llave:valor del diccionario en cuestión.

Nota para tener en cuenta: es muy importante tener en cuenta la sangría en los bucles.

Forma correcta: **for** llave, valor **in** diccionario.items():
expresión

Forma incorrecta: **for** llave, valor **in** diccionario.items():
expresión

Bucles while

Los bucles while son útiles cuando no se tiene idea de cuántas veces necesitas que tu programa itere, como, por ejemplo, un juego de azar. El diagrama de abajo explica el funcionamiento de un bucle while.

Sintaxis: **while** expresión:
Declaración(s)

¿Qué es una lista por comprensión en Python?

Configura un número de bucles for/in para que funcionen en una sola línea y genere listas de esas líneas de código. Las listas por comprensión reducen el código a una sola línea y genera de forma dinámica una nueva lista a partir de una lista preexistente.

Sintaxis: newList = [expresión **for** elemento **in** oldList **if**
condición]

Normalmente estas listas no las usamos ni creamos nosotros. Pero aparecen en muchas bibliotecas, por lo que está bien conocerlas.

¿Qué es un argumento en Python?

Los argumentos son los valores que se pasan dentro del paréntesis de la función. Una función puede tener cualquier número de argumentos separados por una coma.

Existen diferentes tipos de argumentos:

- Argumentos posicionales: aquellos que no son nombrados y se copian por orden de escritura.

Ejemplo:

```
def saludo(nom):
    print(f'Hi {nom}!')
```

saludo(Juana) → devuelve ¡Hola Juana!

- Argumentos nombrados: aquellos precedidos por un identificador. Todos los tipos de argumentos pueden ser argumentos nombrados, lo cual difiere de otros programas. Además, al no ser posicionales, se pueden poner en el orden que se quieran.

EJEMPLO

- Argumentos con valor por defecto: se les asigna un valor al definir la función, pero se pueden sobrescribir. Son mutables, por lo que no usaremos elementos mutables como listas como argumentos con valor por defecto.

Ejemplo:

```
Def suma(x, y = 0),
    return x + y
```

Al llamar la función, si no se especifica el valor de y, lo tomará como cero. Por ejemplo, suma(3, 0) devolverá 3, y suma(3, 7) devolverá 10.

- Número arbitrario de argumentos
 - Unpacking: Acumular los argumentos posicionales en una tupla: cualquier argumento que le pasemos a la función que no se pueda definir en un argumento preexistente, pasa a esa tupla. Se usa cuando tienes una función que necesita una colección de datos como argumentos, pero no sabes cuántos datos hay. Así, al llamar la función se le pueden pasar la cantidad de argumentos que se quiera.

Las buenas prácticas indican llamar a esta tupla *arg.

Ejemplo:

```
def resta(*args):
    return sum(args)
```

print(suma(5, 6, 34) → devolverá 45

print(1, 5, 6, 4, 3) → devolverá 19

- Argumentos clave, colección de argumentos nombrados que devuelve diccionarios. Lo mismo que el caso anterior, pero para diccionarios.

Las buenas prácticas indican llamar a estos diccionarios `**kwargs`

Cuando tenemos muchos argumentos, una buena práctica es ponerlos en una lista.

Ejemplo:

```
saludo('arg1',  
      'arg2',  
      'arg3',  
      'arg4',  
      'arg5'  
):
```

¿Qué es una función Lambda en Python?

Una función Lambda se define como una función anónima. Lambda no es un nombre, es una palabra clave de Python que engloba “lo que viene a continuación es una función anónima.

Sintaxis: `Lambda input: expresión`

- Input es la lista de argumentos
- La expresión define el valor que devuelve.

Características de las expresiones lambda:

- Son similares a las variables.
- Son usadas cuando necesitas funciones cortas y simples para usar en otras funciones y/o en otras partes del programa.
- Se suelen usar para ordenar y filtrar datos.

No devuelven un valor debido a su anonimato. Para que devuelvan algo, la función lambda se debe almacenar en, por ejemplo, una variable.

Ejemplo de uso de las expresiones lambda:

```
ecuacion = lambda alfa, beta, omega: alfa/4 + beta + 3*omega
```

```
print(ecuacion(2, 8, 5)) → devolverá -6.5
```

¿Qué es un paquete pip?

Literalmente es un sistema de gestión de paquetes para paquetes de Python. Razón de ser: se necesita para comunicar la biblioteca de terceros (módulos o paquetes) con el Core lenguaje de Python.

Estos módulos y paquetes, así como el instalador de pip para diferentes sistemas operativos, se encuentra en la página de PyPI (The Python Package Index). El Python 3 en adelante ya lo tiene instalado.

