Tecnicatura Universitaria en Programación
Programación II

UNIDAD N° 1: Lenguaje Python - Sintaxis y Elementos básicos y avanzados - Anexo

Índice

Cortocircuito

La evaluación de cortocircuito denota la semántica de algunos operadores booleanos en algunos lenguajes de programación en los cuales el segundo argumento no se ejecuta o evalúa si el primer argumento de la función **and** evalúa y el resultado es falso, el valor total tiene que ser falso; y cuando el primer argumento de la función **or** evalúa y el resultado es verdadero, el valor total tiene que ser verdadero

Cuando Python está procesando una expresión lógica, como ***x >= 2 and (x/y) > 2***, evalúa la expresión de izquierda a derecha. Debido a la definición de **and**, si x es menor de 2, la expresión ***x >= 2*** resulta ser falsa, de modo que la expresión completa ya va a resultar falsa, independientemente de si ***(x/y) > 2*** se evalúa como verdadera o falsa.

Cuando Python detecta que no se gana nada evaluando el resto de una expresión lógica, detiene su evaluación y no realiza el cálculo del resto de la expresión. Cuando la evaluación de una expresión lógica se detiene debido a que ya se conoce el valor final, eso es conocido como **cortocircuitar la evaluación.**

Ejemplos de utilización:

```
programa.py > ...
1   x = 5
2   y = 0
3
4   resultado = x/y
5   print(resultado)
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

PS D:\Developer\py\python-practice1> python programa.py
Traceback (most recent call last):
  File "D:\Developer\py\python-practice1\programa.py", line 4, in <module>
    resultado = x/y
                ~^~
ZeroDivisionError: division by zero
```

La tercera instrucción ha fallado porque Python intentó evaluar (x/y) lo cual provoca un runtime error (error en tiempo de ejecución) de división por 0.

```
programa.py > ...
1   x = 5
2   y = 0
3
4   resultado = y!=0 and x/y
5   print(resultado)
```

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL

PS D:\Developer\py\python-practice1> python programa.py
False
```

Valiéndonos de la técnica de cortocircuito, en las segunda opción al evaluar la expresión ***y!=0 and x/y*** no se produce ningún error, ya que Python al ver que el primer operando ***y!=0*** es falso, no sigue ejecutando el resto de la expresión lógica.

Si se llama a una función o método en el lado derecho de una expresión lógica, puede no ejecutarse dependiendo del resultado en el lado izquierdo.

programa.py > ...

```
1  def test():
2      print('function is called')
3      return True
4
5  print(True and test())
6  print(False and test()) #La función no se ejecutará
7  print(True or test()) #La función no se ejecutará
8  print(False or test())
9
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS D:\Developer\py\python-practice1> python programa.py
function is called
True
False
True
function is called
True
```

Valores de retorno con and y or

Tenga en cuenta que ni **and** ni **or** restringe el valor devuelto al el tipo False o True, sino que devuelve el valor del último argumento evaluado. Esto a veces es útil, por ejemplo, para reemplazar por un valor predeterminado una cadena si está vacía.

x and y: La expresión primero evalúa x ; si x es falso, se devuelve su valor; de lo contrario, se evalúa y se devuelve el valor resultante.

x or y: La expresión primero evalúa x ; si x es verdadero, se devuelve su valor; de lo contrario, se evalúa y se devuelve el valor resultante.

```python
x = 5
y = 1
z = 0

print(x and y)
print(x or y)
print(x and z)
print(z and x)
print(x or z)
print(z or x)
```

```
PS D:\Developer\py\python-practice1> python programa.py
1
5
0
0
5
5
```

Strings

Reemplazar ocurrencias en una cadena

El método **replace()** reemplaza una frase específica con otra frase específica. Se reemplazarán todas las apariciones de la frase especificada, si no se especifica la cantidad de ocurrencias a reemplazar.

Sintaxis replace()
string.replace(oldvalue, newvalue, count)

```python
cadena = 'No hay que ir para atrás ni para darse impulso'
print(cadena)

cadena_resultado = cadena.replace('para','PARA')
print(cadena_resultado)
```

```
PS C:\PII\py\python-practice1> python program.py
No hay que ir para atrás ni para darse impulso
No hay que ir PARA atrás ni PARA darse impulso
```

program.py > ...

```
1   cadena = 'No hay que ir para atrás ni para darse impulso'
2   print(cadena)
3
4   cadena_resultado = cadena.replace('para','PARA',1)
5   print(cadena_resultado)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\PII\py\python-practice1> python program.py
No hay que ir para atrás ni para darse impulso
No hay que ir PARA atrás ni para darse impulso
```

Tuplas y listas

Desempaquetado

En algunas ocasiones, nos podemos encontrar con una tupla, o lista, que contiene varios valores de los cuales solamente nos interesan unos pocos. Por lo que extraer solamente estos valores y quedarnos con los necesarios puede simplificar los posteriores trabajos. Esto es algo que se puede conseguir mediante el **desempaquetado** en Python de una tupla o lista. Una tarea más sencilla de lo que parece.

El desempaquetado de todos los valores de una tupla o lista se realiza como sigue

program.py > ...

```
1   lista = ["primer", 25, [1, 2, 3]]
2   a, b, c = lista
```

A la hora de desempaquetar en Python una tupla o lista hay que tener en cuenta que es necesario indicar tantas variables como elementos de la lista. En caso contrario se producirá un error.

program.py > ...

```
1   lista = ["primer", 25, [1, 2, 3]]
2   a, b = lista
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\PII\py\python-practice1> python program.py
Traceback (most recent call last):
  File "C:\PII\py\python-practice1\program.py", line 2, in <module>
    a, b = lista
    ^^^^
ValueError: too many values to unpack (expected 2)
```

```
program.py > d
1   lista = ["primer", 25, [1, 2, 3]]
2   a, b, c, d = lista

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\PII\py\python-practice1> python program.py
Traceback (most recent call last):
  File "C:\PII\py\python-practice1\program.py", line 2, in <module>
    a, b, c, d = lista
    ^^^^^^^^^^
ValueError: not enough values to unpack (expected 4, got 3)
```

Para trabajar sólo con alguno valores de la lista que nos interesa, por ejemplo el primero y último elemento, no valemos del * para desempaquetar todos los valores que no nos interesa en una variable.

```
program.py > ...
1   tupla = ("Juan", 25, [1, 2, 3], 10.5)
2   primer_elemento, *_, ultimo_elemento = tupla
3
4   print(ultimo_elemento)

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\PII\py\python-practice1> python program.py
10.5
```

Método extend

el método **extend( )**, a diferencia de **append( )**, itera sobre el elemento que desea agregar, es decir, involucra el argumento pasado por parámetro dentro de un ciclo, y luego agrega los valores contenidos dentro de ese parámetro, es decir, el argumento que se pasa debe ser un iterable cómo por ejemplo otra lista, una tupla, una cadena, ….

```
program.py > ...
1   palabras = []
2
3   palabras.extend("chocolate")
4
5   print(palabras)

PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\PII\py\python-practice1> python program.py
['c', 'h', 'o', 'c', 'o', 'l', 'a', 't', 'e']
```

```
program.py > ...
1   palabras = []
2
3   palabras.extend(['b','a','n','a','n','a'])
4
5   print(palabras)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\PII\py\python-practice1> python program.py
['b', 'a', 'n', 'a', 'n', 'a']
```

```
program.py > ...
1   palabras = []
2
3   palabras.extend(('m','a','n','z','a','n','a'))
4
5   print(palabras)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\PII\py\python-practice1> python program.py
['m', 'a', 'n', 'z', 'a', 'n', 'a']
```

Método reverse

reverse( ) es un método incorporado en el lenguaje de programación Python que invierte los objetos de la Lista en su lugar, es decir, no utiliza ningún espacio adicional sino que simplemente modifica la lista original.

```
program.py > ...
1   list_1 = list(range(1,6))
2   print(list_1)
3   list_1.reverse()
4   print(list_1)
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

PS C:\PII\py\python-practice1> python program.py
[1, 2, 3, 4, 5]
[5, 4, 3, 2, 1]
```

```python
# Dada una palabra verifique si es un palíndromo
palabra = "reconocer"
lista_palabra_reverse = list(palabra)
lista_palabra_reverse.reverse()
palabra_reverse = ''.join(lista_palabra_reverse)

if palabra_reverse == palabra:
    print(palabra, ": Es un palíndromo")
else:
    print(palabra, ": No es un palíndromo")
```

```
PS C:\PII\py\python-practice1> python program.py
reconocer : Es un palíndromo
```

Expresiones Lambda

Las expresiones lambda se usan idealmente cuando necesitamos hacer algo simple y estamos más interesados en hacer el trabajo rápidamente en lugar de nombrar formalmente la función.
Las expresiones lambda también se conocen como funciones anónimas.
Las expresiones lambda en Python son una forma corta de declarar funciones pequeñas y anónimas (no es necesario proporcionar un nombre para las funciones lambda).
Las funciones Lambda se comportan como funciones normales declaradas con la palabra clave def. Resultan útiles cuando se desea definir una función pequeña de forma concisa. Pueden contener sólo una expresión, por lo que no son las más adecuadas para funciones con instrucciones de flujo de control.

Sintaxis Lambda
lambda argumentos: expresión

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

```python
# Función Lambda para calcular el cuadrado de un número
square = lambda x: x ** 2
print(square(3)) # Resultado: 9
```

```
PS C:\PII\py\python-practice1> python program.py
9
```

En el ejemplo de anterior, ***lambda x: x ** 2*** produce un objeto de función anónimo que se puede asociar con cualquier nombre. Entonces, asociamos el objeto de función con la variable ***square***. De ahora en adelante, podemos llamar a la función lambda cómo cualquier función tradicional, por ejemplo, ***square(10)***

Supongamos que desea filtrar los números impares de una lista. Podríamos hacerlo con una función lambda o con un for cómo sigue:

program.py > ...

```
1   mi_lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
2   filtrado = []
3
4   for num in mi_lista:
5       if num % 2 != 0:
6           filtrado.append(num)
7
8   print(filtrado)
```

program.py > ...

```
1   filtrado = [x for x in range(1,11) if x % 2 != 0]
2
3   print(filtrado)
```

Supongamos que tenemos que ordenar una lista ignorando los carácteres mayúsculas o minúsculas, podríamos hacerlo con una función lambda como sigue:

program.py > ...

```
1   nombres = ["MIGUEL", "Marta", "analisa", "julia", "Barbara"]
2   nombres.sort()
3   print(nombres)
4
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\PII\py\python-practice1> python program.py
['Barbara', 'MIGUEL', 'Marta', 'analisa', 'julia']
```

program.py > ...

```python
nombres = ["MIGUEL", "Marta", "analisa", "julia", "Barbara"]
nombres.sort(key=lambda x: x.lower())
print(nombres)
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
PS C:\PII\py\python-practice1> python program.py
['analisa', 'Barbara', 'julia', 'Marta', 'MIGUEL']
```

Bibliografía

https://www.python.org/doc/
Matthes, E. (2023). Python crash course: A hands-on, project-based introduction to programming. no starch press.
https://www.freecodecamp.org/espanol/news/expresiones-lambda-en-python/

Versiones

Fecha	Versión
20/08/2023	1.0

Autores

María Mercedes Valoni