

Sumario

UT 04: Estructuras de control en Python.....	2
1 Bloques.....	2
2 Bifurcaciones.....	2
2.1 Condicionales "if".....	2
2.2 Condiciones múltiples.....	3
3 Bucles.....	4
3.1 While.....	4
3.2 Bucles "for".....	6
4 Resumen de estructuras de programación.....	9
5 Depuración de programas.....	10



Documentación oficial de Python: <https://docs.python.org/3/>

Referencia **w3schools**: https://www.w3schools.com/python/python_reference.asp



Realizado bajo licencia [Creative Commons Reconocimiento-NoComercial CC-BY-NC 4.0](https://creativecommons.org/licenses/by-nc/4.0/)

UT 04: Estructuras de control en Python

1 Bloques

Una de las características más peculiares de Python es que los bloques de programación vienen dados por párrafos con sangrías. A diferencia de otros lenguajes, en que las sangrías y los espacios solo tienen una funcionalidad estética, en Python sí se tiene en cuenta dónde están escritas las instrucciones, por lo que hay que ser muy cuidadoso con la estructura del código.

Los bloques vienen marcados por el carácter ":" y no se pueden dejar vacíos. La instrucción "pass" indica que un bloque no hace nada.

```
inicio del bloque:
    pass
else:
    pass
```

2 Bifurcaciones

2.1 Condicionales "if"

Las estructuras condicionales realizan bloques de código cuando se produce una condición determinada, generalmente expresada con operadores de comparación,

```
if condicion:
    loquehagoisecumple
    y más instrucciones
    en líneas consecutivas
```

```
if condicion:
    loquehagoisecumple
else:
    loquehagoisNOsecumple
    etc...
```

```
if condicion:
    loquehagoisecumplecond
    etc...
elif condicion2:
    loquehagoisecumplec2
    etc...
else:
    loquehagoisNOsecumplen
```

Por ejemplo, veamos un programa para calcular el valor máximo de una serie de números introducidos por el usuario:

```
# read three numbers
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))
number3 = int(input("Enter the third number: "))
# check which one of the numbers is the greatest
# and pass it to the largest_number variable
largest_number=number1
if number2>largest_number:
    largest_number=number2
if number3>largest_number:
    largest_number=number3

largest_number = max(number1, number2, number3)
# print the result
print("The largest number is:", largest_number)
```

A veces encontramos funciones predefinidas en Python, que nos ayudan a realizar el trabajo más rápido. Una alternativa al programa anterior sería:

```
# read three numbers
number1 = int(input("Enter the first number: "))
number2 = int(input("Enter the second number: "))
number3 = int(input("Enter the third number: "))

# check which one of the numbers is the greatest
# and pass it to the largest_number variable
largest_number = max(number1, number2, number3)

# print the result
print("The largest number is:", largest_number)
```

2.2 Condiciones múltiples

En muchos lenguajes de programación se usa la instrucción "switch" (o "case", según el lenguaje) para evaluar una expresión y adoptar soluciones diferentes en función del resultado. En Python no tenemos esa instrucción, pero podemos utilizar otros recursos, como las estructuras de datos denominadas "diccionarios", que veremos más adelante, o bien los condicionales anidados. Por ejemplo:

```
if x > 10:
    #Haz algo...
    if x > 20:
        #Haz algo mas especial...
```

3 Bucles

Los bucles son **secuencias de instrucciones** que se repiten siempre que se den las condiciones oportunas.

El bucle "while" se repite mientras la condición expresada al principio se siga cumpliendo.

El bucle "for" se repite un número determinado de veces, controlado generalmente por una variable de tipo "contador", cuyo valor cambia en cada repetición.

3.1 While

```
while condicion:
    loquehagomientrasecumple
else:
    LoquehagocuandoNOsecumpla
```

En ocasiones, cuando la condición se da de forma permanente, se producen bucles infinitos. Puede ser un comportamiento diseñado por el programador o puede que, por error de programación, la condición de salida no llegue a darse nunca.

Este ejemplo sería un **bucle infinito**:

```
while True:
    print("I'm stuck inside a loop.")
```

Y este sería un bucle **controlado por variable**:

```
i = 0
while i < 100:
    # do_something()
    i += 1
```

Veamos programa para calcular el número más alto de los introducidos por el usuario, hasta que el usuario introduce "-1":

```
# we will store the current largest number here
largest_number = -999999999

# input the first value
number = int(input("Enter a number or type -1 to stop: "))

# if the number is not equal to -1, we will continue
while number != -1:
    # is number larger than largest_number?
    if number > largest_number:
        # yes, update largest_number
        largest_number = number
```

```
# input the next number
number = int(input("Enter a number or type -1 to stop: "))

# print the largest number
print("The largest number is:", largest_number)
```

Otro ejemplo: un programa que calcula el número de valores pares e impares introducidos por el usuario, hasta que se lee el valor "0".

```
# A program that reads a sequence of numbers
# and counts how many numbers are even and how many are odd.
# The program terminates when zero is entered.

odd_numbers = 0
even_numbers = 0
# read the first number
number = int(input("Enter a number or type 0 to stop: "))

# 0 terminates execution
while number != 0:
    # check if the number is odd
    if number % 2 == 1:
        # increase the odd_numbers counter
        odd_numbers += 1
    else:
        # increase the even_numbers counter
        even_numbers += 1
    # read the next number
    number = int(input("Enter a number or type 0 to stop: "))

# print results
print("Odd numbers count:", odd_numbers)
print("Even numbers count:", even_numbers)
```

Para salir de un bucle infinito se puede utilizar la instrucción "break".

Ejercicio: número secreto

Veamos un programa que calcula un número secreto, entre 0 y 100.

Para salir del programa se ejecuta una sentencia "break" cuando se da una cierta condición.

```
secret_number = 777

# A continuación veremos el uso de tres comillas para
# formatear un texto en varias líneas.
```

```
print(
    """
+=====+
| Welcome to my game, muggle! |
| Enter an integer number    |
| and guess what number I've |
| picked for you.           |
| So, what is the secret number? |
+=====+
    """)

intentos=0
while True:
    print("Introduce un número: ")
    guess_number=int(input())
    intentos+=1
    if guess_number==secret_number:
        break
    elif guess_number>secret_number:
        print("El número secreto es más bajo.")
    elif guess_number<secret_number:
        print("El número secreto es más alto.")

print("Acertaste en ",intentos," intentos.")
```

3.2 Bucles "for"

Los bucles "for" están controlados por el valor de una variable, generalmente un contador.

```
for i in range(max):
    loquehagomaxveces

for i in range(max):
    loquehagomaxveces
else:
    Loquehagocuandoacabe
```

`range()` genera un rango

- Con un argumento el máximo
- Con dos mínimo y máximo
- Con tres mín, máx e incremento

Escapar del bucle:

- `break` – sale del bucle
- `continue` – salta a la siguiente iteración del bucle

Ejemplos:

```
#Utilizando una variable autoincrementada por defecto, comenzando por 0
for i in range(100):
    print("The value of i is currently",i)
    pass
```

Salida:

```
The value of i is currently 0
The value of i is currently 1
The value of i is currently 2
The value of i is currently 3
The value of i is currently 4
The value of i is currently 5
The value of i is currently 6
The value of i is currently 7
The value of i is currently 8
The value of i is currently 9
```

```
#Recorriendo la variable i con: comienzo, final e incremento (salto)
for i in range(2, 8, 3):
    print("The value of i is currently", i)
```

Salida:

```
The value of i is currently 2
The value of i is currently 5
```

Hay que prestar atención al uso de "range" con parámetros, porque la secuencia de valores nunca alcanzará este valor máximo. Así, el siguiente ejemplo no generará ninguna salida:

```
for i in range(1,1):
    print("The value of i is currently", i)
```

Además, el incremento siempre se toma como valor positivo (aunque podemos expresarlo en negativo para recorrer los valores en sentido inverso):

```
for i in range(10,1,-2):
    print("The value of i is currently", i)
```

Salida:

```
The value of i is currently 10
The value of i is currently 8
The value of i is currently 6
The value of i is currently 4
The value of i is currently 2
```

Para introducir retrasos en el código se puede usar la función "sleep" (previamente importada de la librería "time"). Por ejemplo, veamos un programa de cuenta atrás:

```
import time
for i in range(10,0,-1):
    print(i)
    time.sleep(1)
print("Launching!!")
```

Ahora veamos unos ejemplos de cómo escapar del bucle con "break" y "continue":

```
# break - example
print("The break instruction:")
for i in range(1, 6):
    if i == 3:
        break
    print("Inside the loop.", i)
print("Outside the loop.")
```

Salida:

```
The break instruction:
Inside the loop. 1
Inside the loop. 2
Outside the loop.
```

```
# continue - example
print("\nThe continue instruction:")
for i in range(1, 6):
    if i == 3:
        continue
    print("Inside the loop.", i)
print("Outside the loop.")
```

Salida:

```
The continue instruction:
Inside the loop. 1
Inside the loop. 2
Inside the loop. 4
Inside the loop. 5
Outside the loop.
```

Como se puede apreciar, la sentencia "break" interrumpe la ejecución del bucle, mientras que la instrucción "continue" solo se salta una de las iteraciones y continúa con la siguiente.

También se puede utilizar "else" con los bucles "for":

```
for i in range(5):
    print(i)
else:
    print("else:", i)
```


4 Resumen de estructuras de programación

Resumen de bucles (del curso oficial de Cisco NetAcad):

1. There are two types of loops in Python: while and for: the while loop executes a statement or a set of statements as long as a specified boolean condition is true, e.g.:

```
# Example 1
while True:
    print("Stuck in an infinite loop.")
```

```
# Example 2
counter = 5
while counter > 2:
    print(counter)
    counter -= 1
```

The for loop executes a set of statements many times; it's used to iterate over a sequence (e.g., a list, a dictionary, a tuple, or a set - you will learn about them soon) or other objects that are iterable (e.g., strings). You can use the for loop to iterate over a sequence of numbers using the built-in range function. Look at the examples below:

```
# Example 1
word = "Python"
for letter in word:
    print(letter, end="*")
```

```
# Example 2
for i in range(1, 10):
    if i % 2 == 0:
        print(i)
```

2. You can use the break and continue statements to change the flow of a loop:

You use break to exit a loop or continue to skip de current iteration:

```
text = "OpenEDG Python Institute"
for letter in text:
    if letter == "P":
        break
    print(letter, end="")
```

```
text = "pyxpyxpyx"
for letter in text:
    if letter == "x":
        continue
    print(letter, end="")
```

3. The while and for loops can also have an else clause in Python. The else clause executes after the loop finishes its execution as long as it has not been terminated by break, e.g.:

```
n = 0
while n != 3:
    print(n)
    n += 1
else:
    print(n, "else")
```

4. The range() function generates a sequence of numbers. It accepts integers and returns range objects. The syntax of range() looks as follows: range(start, stop, step), where:

- start is an optional parameter specifying the starting number of the sequence (0 by default)
- stop is an optional parameter specifying the end of the sequence generated (it is not included),
- and step is an optional parameter specifying the difference between the numbers in the sequence (1 by default.)

Example code:

```
for i in range(3):
    print(i, end=" ") # outputs: 0 1 2

for i in range(6, 1, -2):
    print(i, end=" ") # outputs: 6, 4, 2
```

5 Depuración de programas

Depurar el código de un programa consiste en encontrar posibles errores mediante la técnica de introducir "breakpoints" o puntos de ruptura. Al llegar a estos puntos, la ejecución se detiene de manera que el programador puede observar los valores que tienen las diferentes variables en ese momento. Así se puede contrastar si el programa está funcionando correctamente.

Supongamos que tenemos un programa que no funciona:

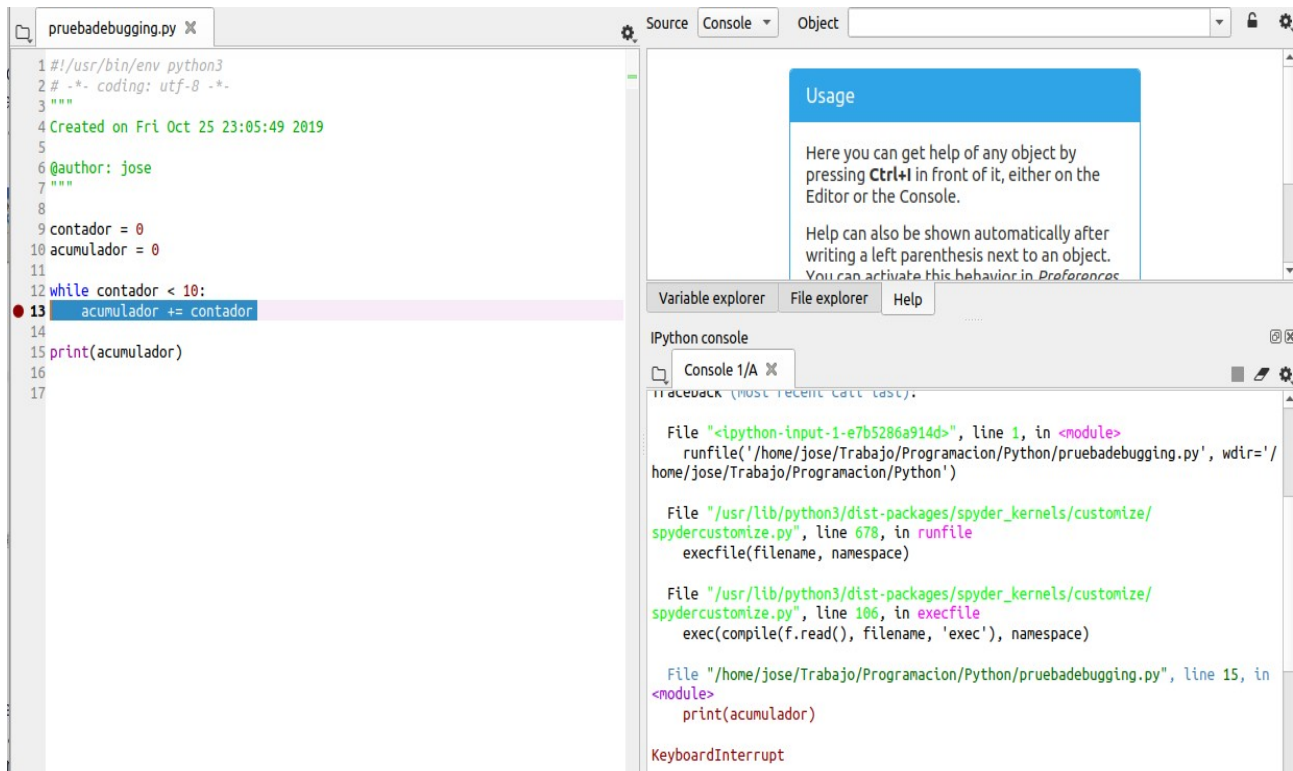
```
contador = 0
acumulador = 0

while contador < 10:
    acumulador += contador

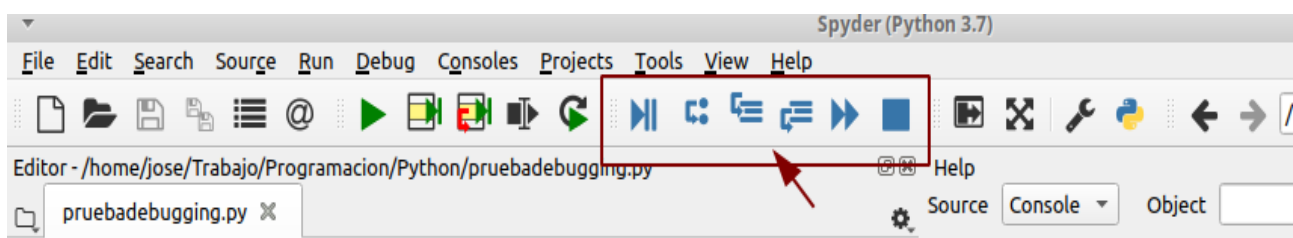
print(acumulador)
```

Dado que el programa no sale nunca del bucle, solo podemos detenerlo mediante la combinación de teclas CTRL+C.

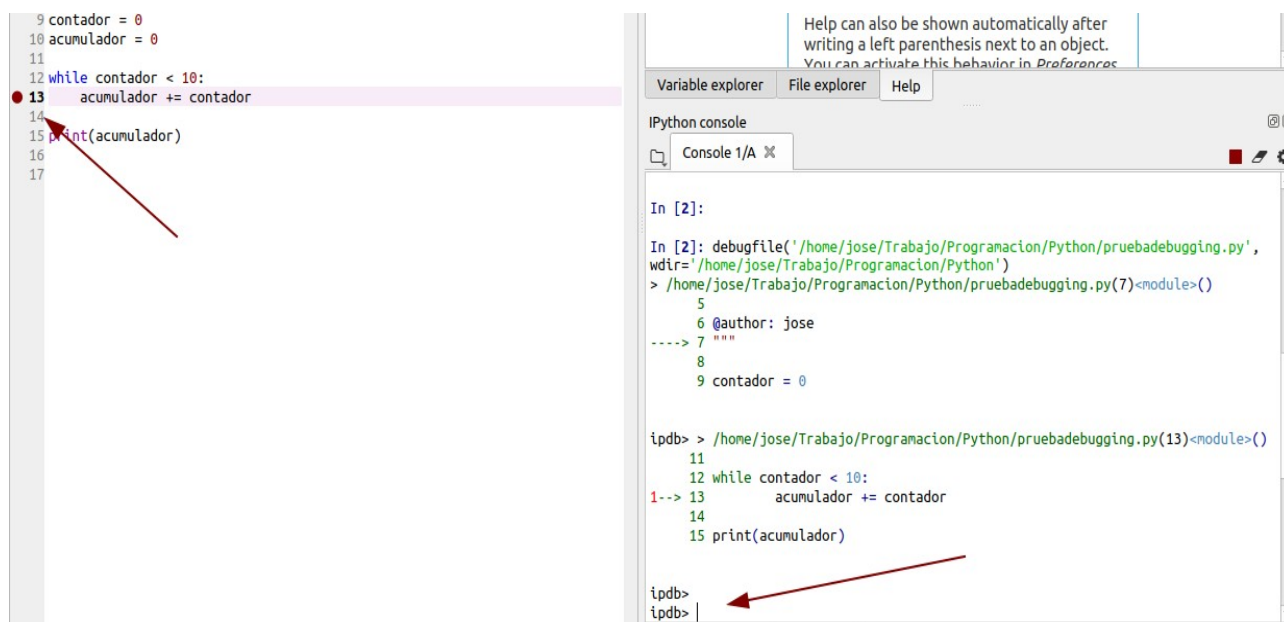
Para investigar qué ocurre, pondremos un punto de ruptura en el interior del bucle, haciendo doble click en el número de línea 13:



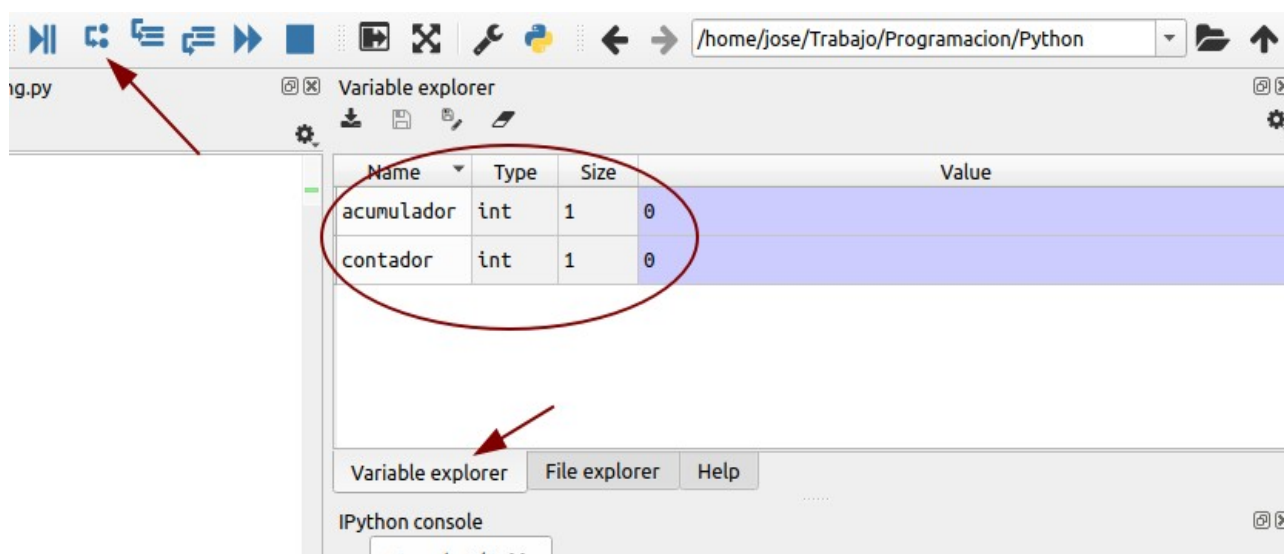
Las funcionalidades del depurador son los botones azules de la parte superior de Spyder:



El botón azul de la izquierda ("Debug file") inicia el proceso de depuración, que se detiene al llegar al punto de ruptura. En ese momento se detiene la ejecución. En consola vemos los pasos ejecutados y nos presenta el prompt del entorno de depuración (`ipdb>`)



En la ventana superior derecha podemos seleccionar la pestaña "Variable explorer", que nos permite ver el valor actual de las variables.



Si pulsamos el botón de ejecución paso a paso, veremos que se van ejecutando sucesivamente las líneas 12 y 13, sin que el flujo de ejecución salga del bucle. En este punto nos daremos cuenta de que se nos ha olvidado codificar la instrucción que permitirá cumplir la condición de salida del bucle.

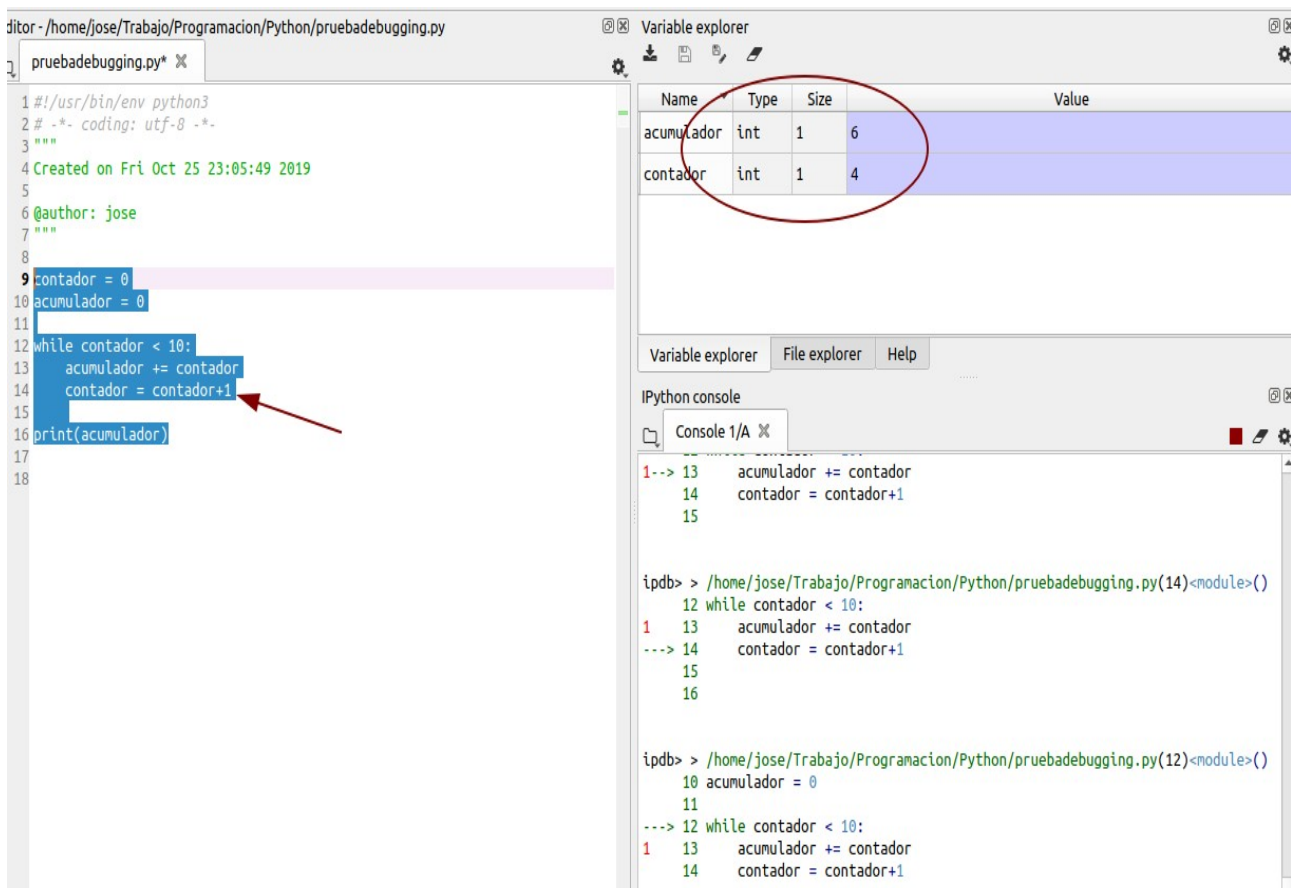
Detenemos el depurador pulsando el cuadrado azul, y corregimos introduciendo la línea que falta al final del bucle:

```
contador = 0
acumulador = 0

while contador < 10:
    acumulador += contador
    contador = contador+1

print(acumulador)
```

Si repetimos el proceso de depuración, ahora veremos que las variables "contador" y "acumulador" van modificando sus valores, es decir, el bucle funciona.



The screenshot displays the Spyder IDE interface. On the left, the editor shows a Python script named `pruebadebugging.py`. The script initializes `contador = 0` and `acumulador = 0`, then enters a `while` loop that increments `acumulador` by `contador` and then increments `contador` by 1, until `contador` reaches 10. The `print(acumulador)` statement is highlighted. A red arrow points to the `contador = contador+1` line. On the right, the Variable explorer shows the current state of the variables: `acumulador` is an integer of size 1 with a value of 6, and `contador` is an integer of size 1 with a value of 4. Below the Variable explorer, the IPython console shows the execution steps, including the `while` loop and the `print` statement.

Name	Type	Size	Value
acumulador	int	1	6
contador	int	1	4

Este ha sido un ejemplo de depuración con Spyder. También se puede hacer depuración en línea de comandos con la interfaz "pdb".