Reducción de datos CCD con IRAF y Python

Alexis Andrés

23 de agosto de 2024

Índice general

Clase 3 | Control de flujo y lógica

3. Controles de flujo

Los controles de flujo en un lenguaje de programación permiten que ciertas partes de un código se ejecuten y otras no, dependiendo de si se cumplen o no algunas condiciones. Adicionalmente, también permiten ejecutar líneas de código una y otra vez mientras alguna condición siga siendo válida. Debido a esto, los principales controles de flujo y lógica son llamados *condicionales* y *bucles* (en inglés, *loops*). En esta clase revisaremos ambos conceptos.

3.1. Condicionales

Son el tipo de controladores de flujo más sencillos, su funcionamiento puede describirse de la siguiente manera: ”Si la variable x es verdadera, entonces realiza una tarea; en caso contrario, realiza esta otra tarea”. Para escribir este tipo de condiciones se utilizan las palabras en inglés «if» y «else», que significan «si» y «en caso contrario», respectivamente.

3.1.1. Declaraciones if

La forma más sencilla de usar los condicionales es cuando solo se quiere comprobar si alguna condición es verdadera y no hacer nada si la condición es falsa. En ese caso, se utiliza la palabra clave `if` una única vez. La sintaxis es la siguiente:

```
if <condition>:
    <if-block>
```

Para este tipo de declaraciones, la condición, identificada como `<condition>` puede ser cualquier tipo de expresión de comparación o lógica que devuelva un valor `True` o `False`. Además, inmediatamente después de dicha condición se escriben dos puntos (`:`). El segmento `<if-block>` representa aquellos comandos o instrucciones que se ejecutarán únicamente si la condición es verdadera.

Algo muy importante es que los comandos del `<if-block>` deberán ser escritos abajo del «`if <condition>:`» con una sangría (en inglés «*indent*») de cuatro espacios. En Python, esta sangría es obligatoria siempre que se quiera separar bloques de código del resto de líneas de código. De no usar la sangría de cuatro espacios en los condicionales, el código generará un error. Como ejemplo, intenta ejecutar la siguiente celda de código:

[1]:
```
#- Definiendo un número
number = 0
if number == 0.0:
print(f"El número {number} es igual a 0.0")
```

```
  Cell In[1], line 3
    print(f"El número {number} es igual a 0.0")
    ^
IndentationError: expected an indented block after 'if' statement on
line 2
```

Este error es bastante simple. Python nos está diciendo que se esperaba un bloque con sangría después de la declaración **`if`**. Puedes corregirlo simplemente agregando la sangría de cuatro espacios:

[2]:
```
#- Definiendo un número
number = 0
if number == 0.0:
    print(f"El número {number} es igual a 0.0")
```

[2]:
```
El número 0 es igual a 0.0
```

La condición `number == 0.0` es verdadera y por lo tanto se mostró el mensaje. Si la condición hubiera sido falsa, no se habría mostrado ningún mensaje. Nota cómo se definió a la variable `number` como de tipo **`int`** y en la condición se comparó con una variable de tipo **`float`**, de modo que la condición sería equivalente a `0 == 0.0`. Esta condición resulta ser cierta porque el operador `==` únicamente compara si dos valores son equivalentes.

3.1.2. Declaraciones if-else

Cada bloque **`if`** puede ir seguido de un bloque opcional **`else`**, que se ejecutará únicamente si la primera condición es falsa. El bloque **`else`** no necesita de ninguna condición pero sí de los dos puntos (`:`) y la línea siguiente también debe tener sangría de cuatro espacios. La sintaxis para este tipo de declaraciones es la siguiente:

```
if <condition>:
    <if-block>

else:
    <else-block>
```

Por ejemplo, podemos escribir un código que verifique si un número es par o impar, comprobando si el número es divisible por 2. Esto se puede hacer con una declaración **`if-else`** de la siguiente manera:

[3]:
```
#- Definiendo un número
number = 15

#- Comprobando si es par o impar
if number % 2 == 0:
    result = "par"
else:
    result = "impar"

# Muestra un mensaje con el resultado
print(f"El número {number} es {result}.")
```

```
El número 15 es impar.
```

El número 15 no es divisible entre dos, por lo tanto la condición `number % 2 == 0` es falsa. Como resultado el bloque `result = "par"` no se ejecuta. En cambio, se ejecuta el bloque `result = "impar"` y se muestra el mensaje `"El número 15 es impar"`.

3.1.3. Declaraciones if-elif-else

En muchas ocasiones será necesario comprobar más de dos posibles situaciones, y para esto se utiliza la sintaxis **`if-elif-else`**. La palabra *elif* es una abreviación de «*else if*» en inglés y en las declaraciones pueden haber cuantos bloques **`elif`** sean necesarios. El primer bloque que devuelva un valor **`True`** será ejecutado, y ninguno de los restantes será comprobado ni ejecutado. La sintaxis de este tipo de declaraciones es la siguiente:

```
if <condition>:
    <if-block>

elif <condition1>:
    <elif-block1>
```

```
elif <condition2>:
    <elif-block2>

else:
    <else-block>
```

El siguiente ejemplo muestra cómo usar las declaraciones **if-elif-else** para clasificar a una partícula fundamental de acuerdo a su masa:

[4]:
```
#- Masa de la partícula hipotética
mass = 5.0  # en GeV/c^2

#- Clasificación basada en su masa
if mass < 0.1:
    classification = "Mesón Ligero"

elif 0.1 <= mass < 1.0:
    classification = "Mesón Pesado"

elif 1.0 <= mass < 10.0:
    classification = "Barión"

else:
    classification = "Partícula Exótica"

#- Mostrar su classificación
print(f"La partícula con masa {mass} GeV/c^2 es un {classification}.")
```

[4]:
```
La partícula con masa 5.0 GeV/c^2 es un Barión.
```

Nuevamente, ten en cuenta que en cada bloque **if-elif-else** se neesita colocar una sangría obligatoria.

3.1.4. Expresiones if-else

Para finalizar con los condicionales, es posible aplicar la sintaxis **if-else** en una simple expresión de una línea (sin los bloques de sangría). Se necesita de tres elementos y por lo tanto se le conoce como un operador condicional ternario. La sintaxis es la siguiente:

```
x if <condition> else y
```

Si la condición es verdadera, entonces el resultado de la expresión anterior es x, si es

falsa, el resultado es y. Esta expresión resulta muy útil para asignar valores a una variable con base en alguna condición. Esto se verifica en el siguiente ejemplo.

[5]:
```
#-Definiendo un número
number = -5

#- Comprobar si es positivo o negativo y guardar el resultado
result = "positivo" if number > 0 else "negativo"

#- Mostrar el resultado
print(f"El número {number} es {result}.")
```

[5]:
```
El número -5 es negativo.
```

En este ejemplo, el operador ternario verifica si la variable `number` es mayor que cero. Si de hecho lo es, le asigna el valor `"positivo"` a la variable `result`, en caso contrario le asigna el valor `"negativo"`.

3.2. Bucles

Como se ha visto, los condicionales permiten que los bloques de código con sangría se ejecuten una única vez dependiendo de alguna condición. Los bucles, en cambio, permiten que el mismo bloque se ejecute muchas veces. Los tipos de bucles disponibles en Python son el bucle **`while`**, el bucle **`for`** y algo que es conocido como «contenedores por comprensión».

3.2.1. Bucle while

Los bucles **`while`** se relacionan con los condicionales porque permiten que el bloque de código se siga ejecutando mientras alguna condición sea verdadera. Su sintaxis es muy similar a la de las declaraciones **`if`**:

```
while <condition>:
    <while-block>
```

Para este tipo de declaraciones también se deben aplicar los dos puntos luego de la condición y el bloque de instrucciones también debe tener sangría. La condición es evaluada al inicio de cada iteración y si es verdadera, el bloque se ejecuta. Si la condición es falsa, el bloque es ignorado y el programa continúa.

El ejemplo más simple usando un bucle while consiste en generar una cuenta regresiva comenzando en un número entero arbitrario:

[6]:
```
#- Número inicial
count = 5

#- Cuenta regresiva
while count > 0:
    print(count)
    count -= 1

print("¡Fin!")  # Se muestra cuando el bucle termina
```

```
5
4
3
2
1
¡Fin!
```

En este ejemplo particular, sucede lo siguiente:

- Se inicia con la variable `count` igual a 5
- El bucle while se ejecuta siempre y cuando `count` sea mayor que cero
- En cada iteración se imprime el valor actual de `count` y posteriormente su valor disminuye en una unidad
- Eventualmente, la variable `count` toma el valor de `0` y la condición se vuelve falsa
- Cuando el bucle termina, se imprime el mensaje `"¡Fin!"`

Es muy común cometer errores en los bucles **`while`**, el más frecuente es iniciar por accidente un bucle infinito. Esto sucede cuando la condición evaluada es siempre verdadera. Si en el ejemplo anterior no hubiésemos escrito la instrucción `count -= 1`, la variable `count` hubiera tenido siempre un valor igual a `5` y por lo tanto el bucle nunca habría finalizado. Al iniciar un bucle **`while`** debemos asegurarnos de agregar una línea de código que haga que la condición sea falsa en algún momento. Si por algún motivo inicias un bucle infinito, siempre puedes detenerlo con la combinación de teclas **`Ctrl+C`**.

También es posible terminar con el bucle **`while`** en cualquier momento, haciendo uso de la instrucción **`break`**. Por ejemplo, si quisiéramos encontrar el primer número par de una lista, podríamos hacerlo con el bucle **`while`** de la siguiente manera:

[7]:
```python
#- Lista de números
numbers = [3, 7, 12, 5, 8, 10, 15]

#- Variable para guardar el número par
first_even = None

#- Bucle while
index = 0
while index < len(numbers):
    if numbers[index] % 2 == 0:
        first_even = numbers[index]
        break  # El bucle termina al encontrar el primer número par
    index += 1

#- Imprime el valor encontrado
if first_even is not None:
    print(f"El primer número par encontrado fue {first_even}.")
else:
    print("No hay números pares en la lista.")
```

[7]:
```
El primer número par encontrado fue 12.
```

3.2.2. Bucle for

Aunque los bucles **`while`** son bastante útiles para repetir declaraciones, en general resulta de mayor utilidad el iterar sobre un contenedor (como las listas, tuplas, conjuntos y diccionarios) o algún otro tipo de variable iterable (como las cadenas de caracteres). En estos casos, se toma un elemento del contenedor en cada iteración y el bucle termina cuando ya no hay más elementos. Esto se logra usando un bucle **`for`**. La sintaxis para los bucles **`for`** es diferente a la del bucle **`while`**, ya que no depende de ninguna condición y consta de lo siguiente:

```
for <variable> in <iterable>:
    <for-block>
```

De nuevo, necesitamos escribir los dos puntos para especificar cuándo inicia el bloque de código que se va a repetir, que a su vez debe tener sangría. En la sintaxis anterior, `<variable>` es el nombre de una variable que se asigna a un elemento cada vez que se ejecuta el bucle. El `<iterable>` es cualquier objeto que pueda devolver elementos. Todos los tipos de contenedores vistos en la Clase 2 son iterables. La cuenta regresiva puede reescribirse de la siguiente manera usando el bucle **`for`**:

[8]:
```
for count in [5,4,3,2,1]:
    print(count)

print("¡Fin!")
```

```
5
4
3
2
1
¡Fin!
```

La instrucción **break** también se puede usar con los bucles **for**, y funcionan de la misma manera. Adicionalmente, también es posible utilizar la instrucción **continue**, que ignora el bloque de código únicamente en la iteración actual y luego continua con la siguiente. Por ejemplo, el siguiente ejemplo muestra cómo ignorar los números impares en una lista:

[9]:
```
for num in [1, 2, 3, 4, 5, 6, 7, 8, 9]:
    if num % 2 != 0:
        continue  # Ignorar y continuar con la siguiente iteración
    print(num)

print("¡Listo!")
```

```
2
4
6
8
¡Listo!
```

La instrucción **continue** también se puede utilizar en los bucles **while**.

Para usar un bucle **for** con una cadena de caracteres se procede de la misma forma que con las listas. El resultado es que se itera por cada una de las letras, por ejemplo:

[10]:
```
for letter in "Hola":
    print(letter)
```

```
H
o
l
a
```

También es posible aplicar un bucle **for** en conjuntos y diccionarios, pero recuerda que este tipo de contenedores no está ordenado, así que el resultado podría no verse como esperarías. Para ilustrar esto se muestra el siguiente ejemplo:

[11]:
```
# Iteración sobre un conjunto
for key in {'a', 'b', 'c', 'd', 'e'}:
    print(key)
```

```
b
c
f
d
a
e
```

El orden en que se muestra el resultado no es necesariamente al orden en el que se escribieron las letras `"a"`, `"b"`, `"c"`, `"d"`, `"e"` y `"f"`. Como último ejemplo en esta sección, se muestra cómo puede usarse el ciclo **`for`** para iterar sobre los pares de clave-valor de un diccionario.

[12]:
```
#- Definiendo el diccionario
d = {'a': 1, 'b': 2, 'c': 3}
```

[13]:
```
#- Iterar sobre las claves
print("Keys:")
for key in d.keys():
    print(key)
print("===========")
```

```
Keys:
a
b
c
===========
```

[14]:
```
#- Iterar sobre los valores
print("Values:")
for value in d.values():
    print(value)
print("===========")
```

```
Values:
1
2
3
===========
```

[15]:
```
#Iterar sobre los pares clave-valor
print("Items")
for key, value in d.items():
    print(f"Key: {key}, Value: {value}")
```

```
Items
Key: a, Value: 1
Key: b, Value: 2
Key: c, Value: 3
```

3.2.3. Contenedores por comprensión

Como se ha visto, los ciclos **`while`** y **`for`** permiten hacer muchas cosas, pero para usarlos se necesita de al menos dos líneas de código: una para definir el bucle y la otra para especificar las acciones. Por ejemplo, supongamos que tenemos una lista con los nombres de los primeros cuatro planetas escritos en minúscula, y también tenemos una lista vacía en la que deseamos agregar esos mismos nombres pero en mayúscula:

[16]:
```
#- Lista con planetas
planetas = ['mercurio', 'venus', 'tierra', 'marte']

#- Lista vacía
planetas_mayus = []
```

Para lograrlo podemos aplicar el método `list.upper()` a cada elemento de la lista `planetas` y luego anexarlos a la lista `planetas_mayus` en un bucle **`for`**:

[17]:
```
for planet in planetas:
    planetas_mayus.append(planet.upper())

print(planetas_mayus)
```

```
['MERCURIO', 'VENUS', 'TIERRA', 'MARTE']
```

Lo anterior puede realizarse de manera equivalente en una sola línea gracias a la sintaxis de contenedores por comprensión, que puede aplicarse tanto a listas como a diccionarios. La sintaxis en cada caso es:

```
# Lista por comprensión
[<expr> for <loop-var> in <iterable>]

# Diccionario por comprensión
{<key-expr>: <value-expr> for <loop-var> in <iterable>}
```

Así, volviendo al ejemplo de los planetas, se puede crear la lista de la siguiente manera:

[18]:
```
planetas_mayus = [planet.upper() for planet in planetas]
```

Para el caso de un diccionario, supongamos que tenemos una lista de números y queremos crear una nueva lista con el cuadrado de esos números, entonces:

[19]:
```
#- Definiendo una lista de números
numbers = [1, 10, 12.5, 65, 88]

#- Creando un diccionario por comprensión
results = {x: x**2 for x in numbers}

#- Mostrar el resultado
print(results)
```

```
{1: 1, 10: 100, 12.5: 156.25, 65: 4225, 88: 7744}
```

También es posible realizar filtros a los contenedores por comprensión, con ayuda de condicionales. La sintaxis es la siguiente:

[20]:
```
# Lista por comprensión con filtro
[<expr> for <loop-var> in <iterable> if <condition>]

# Diccionario por comprensión con filtro
{<key-expr>: <value-expr> for <loop-var> in <iterable> if <condition>}
```

En este caso, si la condición resulta ser verdadera, entonces se agrega el elemento actual, en caso contrario se ignora y se evalúa el siguiente iterable. En este último ejemplo, se creará una lista con los planetas escritos en mayúscula si el nombre inicia con «m», y se calcularán los cuadrados de los números si el número es par:

[21]:
```
#- Planetas que inician con M
new_list = [planet.upper() for planet in planetas if planet[0] == 'm']
print(new_list)
```

```
['MERCURIO', 'MARTE']
```

[22]:
```
#- Cuadrado de números pares
results_even = {x: x**2 for x in numbers if x%2 ==0}
print(results_even)
```

```
{10: 100, 88: 7744}
```