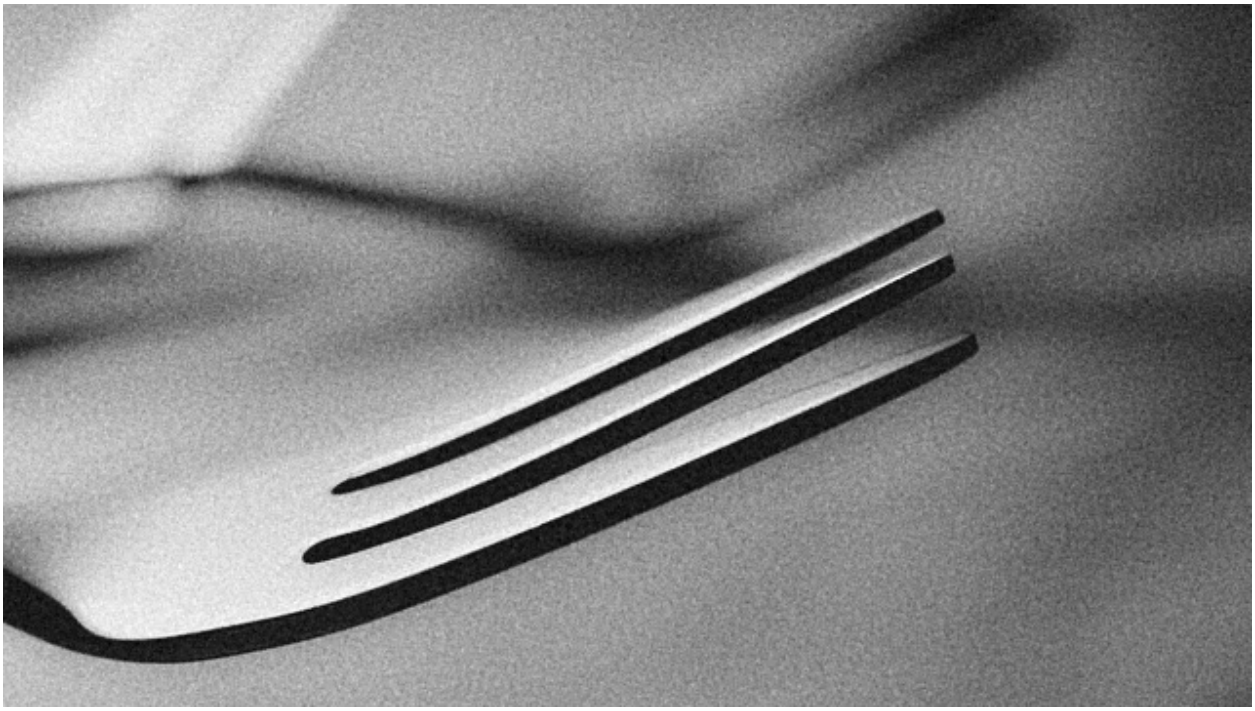
CAPÍTULO 4

Control de flujo

Todo programa informático está formado por *instrucciones* que se ejecutan en forma secuencial de «arriba» a «abajo», de igual manera que leeríamos un libro. Este orden constituye el llamado **flujo** del programa. Es posible modificar este flujo secuencial para que tome *bifurcaciones* o *repita* ciertas instrucciones. Las sentencias que nos permiten hacer estas modificaciones se engloban en el *control de flujo*.

4.1 Condicionales



En esta sección veremos las sentencias `if` y `match-case` junto a las distintas variantes que pueden asumir, pero antes de eso introduciremos algunas cuestiones generales de *escritura de código*.[1]

4.1.1 Definición de bloques

A diferencia de otros lenguajes que utilizan llaves para definir los bloques de código, cuando Guido Van Rossum *creó el lenguaje* quiso evitar estos caracteres por considerarlos innecesarios. Es por ello que en Python los bloques de código se definen a través de **espacios en blanco, preferiblemente 4**.[2] En términos técnicos se habla del **tamaño de indentación**.

Consejo: Esto puede resultar extraño e incómodo a personas que vienen de otros lenguajes de programación pero desaparece rápido y se siente natural a medida que se escribe código.

---

[1] Foto original de portada por ali nafezarefi en Unsplash.
[2] Reglas de indentación definidas en PEP 8

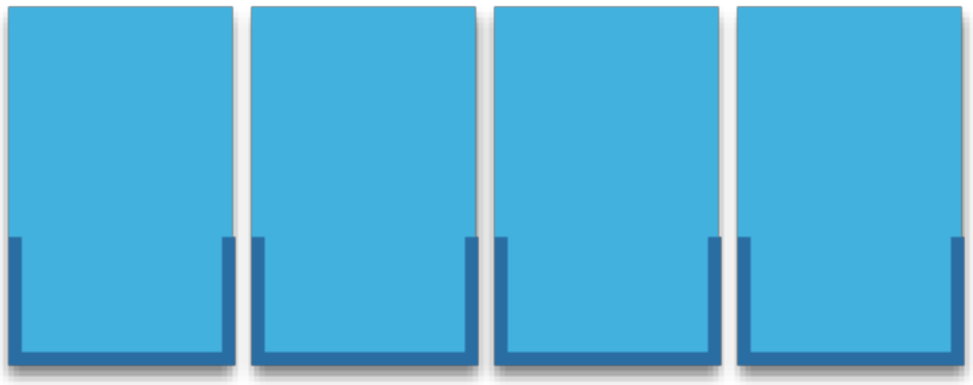


Figura 1: Python recomienda 4 espacios en blanco para indentar

4.1.2 Comentarios

Los comentarios son anotaciones que podemos incluir en nuestro programa y que nos permiten aclarar ciertos aspectos del código. Estas indicaciones son ignoradas por el intérprete de Python.

Los comentarios se incluyen usando el símbolo almohadilla # y comprenden hasta el final de la línea.

Lista 1: Comentario en bloque

```
# Universe age expressed in days
universe_age = 13800 * (10 ** 6) * 365
```

Los comentarios también pueden aparecer en la misma línea de código, aunque la guía de estilo de Python no aconseja usarlos en demasía:

Lista 2: Comentario en línea

```
stock = 0   # Release additional articles
```

Reglas para escribir buenos comentarios:[6]

1. Los comentarios no deberían duplicar el código.
2. Los buenos comentarios no arreglan un código poco claro.
3. Si no puedes escribir un comentario claro, puede haber un problema en el código.
4. Los comentarios deberían evitar la confusión, no crearla.
5. Usa comentarios para explicar código no idiomático.
6. Proporciona enlaces a la fuente original del código copiado.

[6] Referencia: Best practices for writing code comments

7. Incluye enlaces a referencias externas que sean de ayuda.
8. Añade comentarios cuando arregles errores.
9. Usa comentarios para destacar implementaciones incompletas.

4.1.3 Ancho del código

Los programas suelen ser más legibles cuando las líneas no son excesivamente largas. La longitud máxima de línea recomendada por la guía de estilo de Python es de **80 caracteres**.

Sin embargo, esto genera una cierta polémica hoy en día, ya que los tamaños de pantalla han aumentado y las resoluciones son mucho mayores que hace años. Así las líneas de más de 80 caracteres se siguen visualizando correctamente. Hay personas que son más estrictas en este límite y otras más flexibles.

En caso de que queramos **romper una línea de código** demasiado larga, tenemos dos opciones:

1. Usar la *barra invertida* `\`:

```
>>> factorial = 4 * 3 * 2 * 1

>>> factorial = 4 * \
...             3 * \
...             2 * \
...             1
```

2. Usar los *paréntesis* `(...)`:

```
>>> factorial = 4 * 3 * 2 * 1

>>> factorial = (4 *
...              3 *
...              2 *
...              1)
```

4.1.4 La sentencia `if`

La sentencia condicional en Python (al igual que en muchos otros lenguajes de programación) es `if`. En su escritura debemos añadir una **expresión de comparación** terminando con dos puntos al final de la línea. Veamos un ejemplo:

```
>>> temperature = 40
```

(continué en la próxima página)

(proviene de la página anterior)

```
>>> if temperature > 35:
...     print('Aviso por alta temperatura')
...
Aviso por alta temperatura
```

Nota: Nótese que en Python no es necesario incluir paréntesis `(` y `)` al escribir condiciones. Hay veces que es recomendable por claridad o por establecer prioridades.

En el caso anterior se puede ver claramente que la condición se cumple y por tanto se ejecuta la instrucción que tenemos dentro del cuerpo de la condición. Pero podría no ser así. Para controlar ese caso existe la sentencia `else`. Veamos el mismo ejemplo anterior pero añadiendo esta variante:

```
>>> temperature = 20

>>> if temperature > 35:
...     print('Aviso por alta temperatura')
... else:
...     print('Parámetros normales')
...
Parámetros normales
```

Podríamos tener incluso condiciones dentro de condiciones, lo que se viene a llamar técnicamente **condiciones anidadas**[3]. Veamos un ejemplo ampliando el caso anterior:

```
>>> temperature = 28

>>> if temperature < 20:
...     if temperature < 10:
...         print('Nivel azul')
...     else:
...         print('Nivel verde')
... else:
...     if temperature < 30:
...         print('Nivel naranja')
...     else:
...         print('Nivel rojo')
...
Nivel naranja
```

Python nos ofrece una mejora en la escritura de condiciones anidadas cuando aparecen consecutivamente un `else` y un `if`. Podemos sustituirlos por la sentencia `elif`:

[3] El anidamiento (o «nesting») hace referencia a incorporar sentencias unas dentro de otras mediante la inclusión de diversos niveles de profundidad (indentación).

Figura 2: Construcción de la sentencia `elif`

Apliquemos esta mejora al código del ejemplo anterior:

```
>>> temperature = 28

>>> if temperature < 20:
...     if temperature < 10:
...         print('Nivel azul')
...     else:
...         print('Nivel verde')
... elif temperature < 30:
...     print('Nivel naranja')
... else:
...     print('Nivel rojo')
...
Nivel naranja
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/wd58B4t

4.1.5 Operadores de comparación

Cuando escribimos condiciones debemos incluir alguna expresión de comparación. Para usar estas expresiones es fundamental conocer los operadores que nos ofrece Python:

Operador	Símbolo
Igualdad	==
Desigualdad	!=
Menor que	<
Menor o igual que	<=
Mayor que	>
Mayor o igual que	>=

A continuación vamos a ver una serie de ejemplos con expresiones de comparación. Téngase

en cuenta que estas expresiones habría que incluirlas dentro de la sentencia condicional en el caso de que quisiéramos tomar una acción concreta:

```
# Asignación de valor inicial
>>> value = 8

>>> value == 8
True

>>> value != 8
False

>>> value < 12
True

>>> value <= 7
False

>>> value > 4
True

>>> value >= 9
False
```

Podemos escribir condiciones más complejas usando los operadores lógicos:

- and
- or
- not

```
# Asignación de valor inicial
>>> x = 8

>>> x > 4 or x > 12  # True or False
True

>>> x < 4 or x > 12  # False or False
False

>>> x > 4 and x > 12  # True and False
False

>>> x > 4 and x < 12  # True and True
True

>>> not(x != 8)  # not False
```

(continué en la próxima página)

(proviene de la página anterior)

```
True
```

Python ofrece la posibilidad de ver si un valor está entre dos límites de manera directa. Así, por ejemplo, para descubrir si `value` está entre *4* y *12* haríamos:

```
>>> 4 <= value <= 12
True
```

Nota:

1. Una expresión de comparación siempre devuelve un valor *booleano*, es decir `True` o `False`.
2. El uso de paréntesis, en función del caso, puede aclarar la expresión de comparación.

Ejercicio

Dada una variable `year` con un valor entero, compruebe si dicho año es **bisiesto** o no lo es.

ℹ️ Un año es bisiesto en el calendario Gregoriano, si es divisible entre 4 y no divisible entre 100, o bien si es divisible entre 400. Puedes hacer la comprobación en esta lista de años bisiestos.

Ejemplo

- Entrada: `2008`
- Salida: `Es un año bisiesto`

«Booleanos» en condiciones

Cuando queremos preguntar por la **veracidad** de una determinada variable «booleana» en una condición, la primera aproximación que parece razonable es la siguiente:

```
>>> is_cold = True

>>> if is_cold == True:
...     print('Coge chaqueta')
... else:
...     print('Usa camiseta')
...
Coge chaqueta
```

Pero podemos *simplificar* esta condición tal que así:

```
>>> if is_cold:
...     print('Coge chaqueta')
... else:
...     print('Usa camiseta')
...
Coge chaqueta
```

Hemos visto una comparación para un valor «booleano» verdadero (`True`). En el caso de que la comparación fuera para un valor falso lo haríamos así:

```
>>> is_cold = False

>>> if not is_cold:  # Equivalente a if is_cold == False
...     print('Usa camiseta')
... else:
...     print('Coge chaqueta')
...
Usa camiseta
```

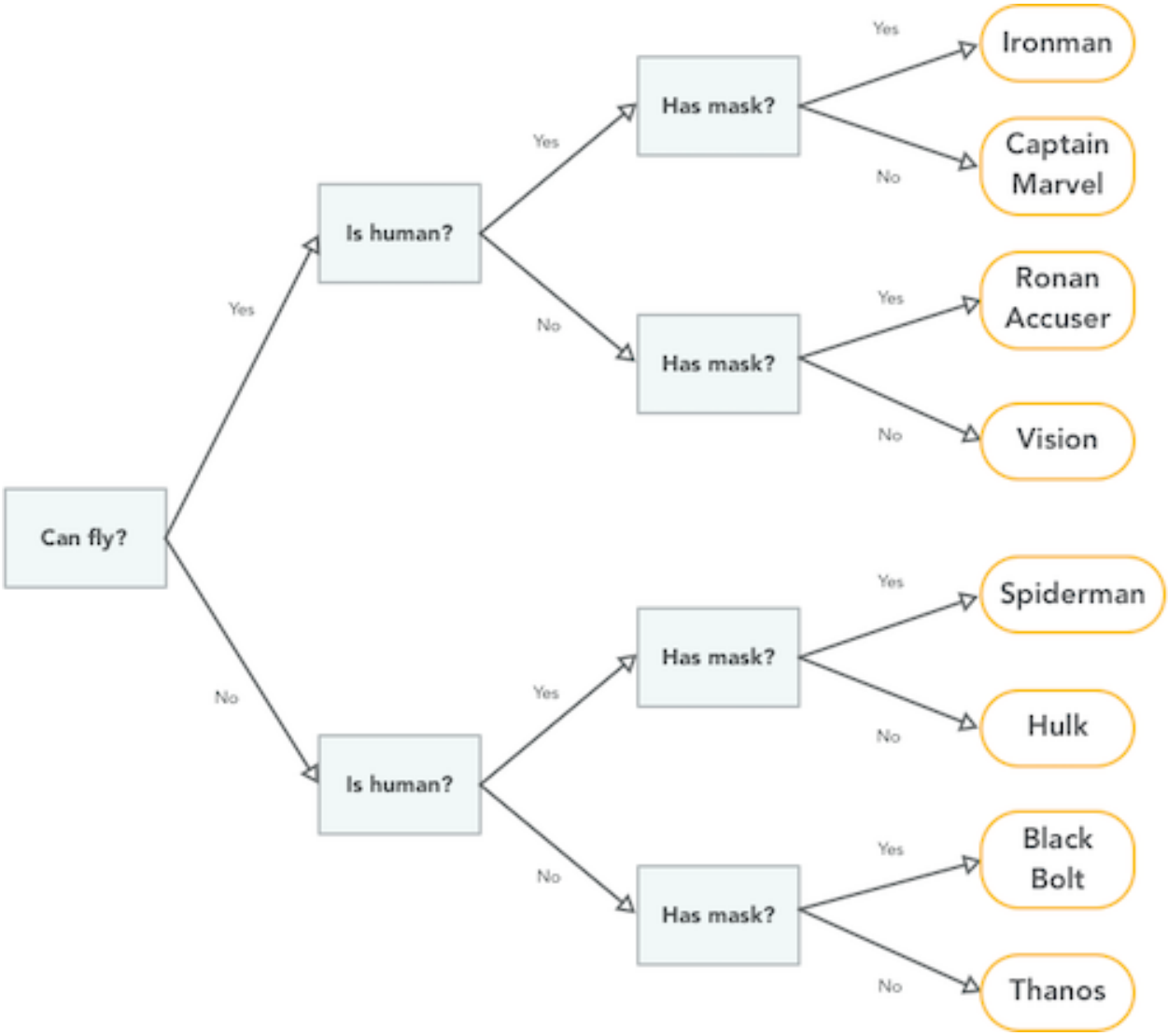
De hecho, si lo pensamos, estamos reproduciendo bastante bien el *lenguaje natural*:

- Si hace frío, coge chaqueta.
- Si no hace frío, usa camiseta.

Ejercicio

Escriba un programa que permita adivinar un personaje de Marvel en base a las tres preguntas siguientes:

1. ¿Puede volar?
2. ¿Es humano?
3. ¿Tiene máscara?



Ejemplo

- Entrada: `can_fly = True`, `is_human = True` y `has_mask = True`
- Salida: `Ironman`

Es una especie de Akinator para personajes de Marvel...

Valor nulo

Nivel intermedio

`None` es un valor especial de Python que almacena el **valor nulo**[4]. Veamos cómo se comporta al incorporarlo en condiciones de veracidad:

```
>>> value = None

>>> if value:
...     print('Value has some useful value')
... else:
...     # value podría contener None, False (u otro)
...     print('Value seems to be void')
...
Value seems to be void
```

Para distinguir `None` de los valores propiamente booleanos, se recomienda el uso del operador `is`. Veamos un ejemplo en el que tratamos de averiguar si un valor **es nulo**:

```
>>> value = None

>>> if value is None:
...     print('Value is clearly None')
... else:
...     # value podría contener True, False (u otro)
...     print('Value has some useful value')
...
Value is clearly void
```

De igual forma, podemos usar esta construcción para el caso contrario. La forma «pitónica» de preguntar si algo **no es nulo** es la siguiente:

```
>>> value = 99

>>> if value is not None:
...     print(f'{value=}')
...
value=99
```

[4] Lo que en otros lenguajes se conoce como `nil`, `null`, `nothing`.

4.1.6 Sentencia `match-case`

Una de las novedades más esperadas (y quizás controvertidas) de Python 3.10 fue el llamado Structural Pattern Matching que introdujo en el lenguaje una nueva sentencia condicional. Ésta se podría asemejar a la sentencia «switch» que ya existe en otros lenguajes de programación.

Comparando valores

En su versión más simple, el «pattern matching» permite comparar un valor de entrada con una serie de literales. Algo así como un conjunto de sentencias «if» encadenadas. Veamos esta aproximación mediante un ejemplo:

```
>>> color = '#FF0000'

>>> match color:
...     case '#FF0000':
...         print('🔴')
...     case '#00FF00':
...         print('🟢')
...     case '#0000FF':
...         print('🔵')
...
🔴
```

¿Qué ocurre si el valor que comparamos no existe entre las opciones disponibles? Pues en principio, nada, ya que este caso no está cubierto. Si lo queremos controlar, hay que añadir una nueva regla utilizando el subguión _ como patrón:

```
>>> color = '#AF549B'

>>> match color:
...     case '#FF0000':
...         print('🔴')
...     case '#00FF00':
...         print('🟢')
...     case '#0000FF':
...         print('🔵')
...     case _:
...         print('Unknown color!')
...
Unknown color!
```

Ejercicio

Escriba un programa en Python que pida (por separado) dos valores numéricos y un operando (suma, resta, multiplicación, división) y calcule el resultado de la operación, usando para ello la sentencia `match-case`.

Controlar que la operación no sea una de las cuatro predefinidas. En este caso dar un mensaje de error y no mostrar resultado final.

Ejemplo

- Entrada: `4`, `3`, `+`
- Salida: `4+3=7`

Patrones avanzados

Nivel avanzado

La sentencia `match-case` va mucho más allá de una simple comparación de valores. Con ella podremos deconstruir estructuras de datos, capturar elementos o mapear valores.

Para ejemplificar varias de sus funcionalidades, vamos a partir de una *tupla* que representará un punto en el plano (2 coordenadas) o en el espacio (3 coordenadas). Lo primero que vamos a hacer es detectar en qué dimensión se encuentra el punto:

```
>>> point = (2, 5)

>>> match point:
...     case (x, y):
...         print(f'({x},{y}) is in plane')
...     case (x, y, z):
...         print(f'({x},{y},{z}) is in space')
...
(2,5) is in plane

>>> point = (3, 1, 7)

>>> match point:
...     case (x, y):
...         print(f'({x},{y}) is in plane')
...     case (x, y, z):
...         print(f'({x},{y},{z}) is in space')
...
(3,1,7) is in space
```

En cualquier caso, esta aproximación permitiría un punto formado por «strings»:

```
>>> point = ('2', '5')

>>> match point:
...     case (x, y):
...         print(f'({x},{y}) is in plane')
...     case (x, y, z):
...         print(f'({x},{y},{z}) is in space')
...
(2,5) is in plane
```

Por lo tanto, en un siguiente paso, podemos restringir nuestros patrones a valores enteros:

```
>>> point = ('2', '5')

>>> match point:
...     case (int(), int()):
...         print(f'{point} is in plane')
...     case (int(), int(), int()):
...         print(f'{point} is in space')
...     case _:
...         print('Unknown!')
...
Unknown!

>>> point = (3, 9, 1)

>>> match point:
...     case (int(), int()):
...         print(f'{point} is in plane')
...     case (int(), int(), int()):
...         print(f'{point} is in space')
...     case _:
...         print('Unknown!')
...
(3, 9, 1) is in space
```

Imaginemos ahora que nos piden calcular la distancia del punto al origen. Debemos tener en cuenta que, a priori, desconocemos si el punto está en el plano o en el espacio:

```
>>> point = (8, 3, 5)

>>> match point:
...     case (int(x), int(y)):
...         dist_to_origin = (x ** 2 + y ** 2) ** (1 / 2)
...     case (int(x), int(y), int(z)):
...         dist_to_origin = (x ** 2 + y ** 2 + z ** 2) ** (1 / 2)
```

(continué en la próxima página)

(proviene de la página anterior)

```
...     case _:
...         print('Unknown!')
...

>>> dist_to_origin
9.899494936611665
```

Con este enfoque, nos aseguramos que los puntos de entrada deben tener todas sus coordenadas como valores enteros:

```
>>> point = ('8', 3, 5)  # Nótese el 8 como "string"

>>> match point:
...     case (int(x), int(y)):
...         dist_to_origin = (x ** 2 + y ** 2) ** (1 / 2)
...     case (int(x), int(y), int(z)):
...         dist_to_origin = (x ** 2 + y ** 2 + z ** 2) ** (1 / 2)
...     case _:
...         print('Unknown!')
...
Unknown!
```

Cambiando de ejemplo, veamos un fragmento de código en el que tenemos que **comprobar la estructura de un bloque de autenticación** definido mediante un *diccionario*. Los métodos válidos de autenticación son únicamente dos: bien usando nombre de usuario y contraseña, o bien usando correo electrónico y «token» de acceso. Además, los valores deben venir en formato cadena de texto:

```
>>> # Lista de diccionarios
>>> auths = [
...     {'username': 'sdelquin', 'password': '1234'},
...     {'email': 'sdelquin@gmail.com', 'token': '4321'},
...     {'email': 'test@test.com', 'password': 'ABCD'},
...     {'username': 'sdelquin', 'password': 1234}
... ]

>>> for auth in auths:
...     print(auth)
...     match auth:
...         case {'username': str(username), 'password': str(password)}:
...             print('Authenticating with username and password')
...             print(f'{username}: {password}')
...         case {'email': str(email), 'token': str(token)}:
...             print('Authenticating with email and token')
...             print(f'{email}: {token}')
```

(continué en la próxima página)

(proviene de la página anterior)

```
...         case _:
...             print('Authenticating method not valid!')
...     print('---')
...
{'username': 'sdelquin', 'password': '1234'}
Authenticating with username and password
sdelquin: 1234
---
{'email': 'sdelquin@gmail.com', 'token': '4321'}
Authenticating with email and token
sdelquin@gmail.com: 4321
---
{'email': 'test@test.com', 'password': 'ABCD'}
Authenticating method not valid!
---
{'username': 'sdelquin', 'password': 1234}
Authenticating method not valid!
---
```

Cambiando de ejemplo, a continuación veremos un código que nos indica si, dada la edad de una persona, puede beber alcohol:

```
>>> age = 21

>>> match age:
...     case 0 | None:
...         print('Not a person')
...     case n if n < 17:
...         print('Nope')
...     case n if n < 22:
...         print('Not in the US')
...     case _:
...         print('Yes')
...
Not in the US
```

- En la **línea 4** podemos observar el uso del operador **OR**.
- En las **líneas 6 y 8** podemos observar el uso de condiciones dando lugar a **cláusulas guarda**.

4.1.7 Operador morsa

Nivel avanzado

A partir de Python 3.8 se incorpora el operador morsa[5] que permite unificar **sentencias de asignación dentro de expresiones**. Su nombre proviene de la forma que adquiere `:=`

Supongamos un ejemplo en el que computamos el perímetro de una circunferencia, indicando al usuario que debe incrementarlo siempre y cuando no llegue a un mínimo establecido.

Versión tradicional

```
>>> radius = 4.25
... perimeter = 2 * 3.14 * radius
... if perimeter < 100:
...     print('Increase radius to reach minimum perimeter')
...     print('Actual perimeter: ', perimeter)
...
Increase radius to reach minimum perimeter
Actual perimeter:  26.69
```

Versión con operador morsa

```
>>> radius = 4.25
... if (perimeter := 2 * 3.14 * radius) < 100:
...     print('Increase radius to reach minimum perimeter')
...     print('Actual perimeter: ', perimeter)
...
Increase radius to reach minimum perimeter
Actual perimeter:  26.69
```

Consejo: Como hemos comprobado, el operador morsa permite realizar asignaciones dentro de expresiones, lo que, en muchas ocasiones, permite obtener un código más compacto. Sería conveniente encontrar un equilibrio entre la expresividad y la legibilidad.

---

[5] Se denomina así porque el operador `:=` tiene similitud con los colmillos de una morsa.

EJERCICIOS DE REPASO

1. Escriba un programa en Python que acepte la opción de dos jugadoras en Piedra-Papel-Tijera y decida el resultado (`solución`).

   Entrada: persona1=piedra; persona2=papel
   Salida: Gana persona2: El papel envuelve a la piedra

2. Escriba un programa en Python que acepte 3 números y calcule el mínimo (`solución`).

   Entrada: 7, 4, 9
   Salida: 4

3. Escriba un programa en Python que acepte un país (como «string») y muestre por pantalla su bandera (como «emoji»). *Puede restringirlo a un conjunto limitado de países* (`solución`).

   Entrada: Italia
   Salida:

4. Escriba un programa en Python que acepte 3 códigos de teclas y muestre por pantalla la acción que se lleva a cabo en sistemas Ubuntu Linux (`solución`).

   Entrada: tecla1=Ctrl; tecla2=Alt; tecla3=Del;
   Salida: Log out

5. Escriba un programa en Python que acepte edad, peso, pulso y plaquetas, y determine si una persona cumple con estos requisitos para donar sangre.

   Entrada: edad=34; peso=81; heartbeat=70; plaquetas=150000
   Salida: Apto para donar sangre

AMPLIAR CONOCIMIENTOS

- How to Use the Python or Operator
- Conditional Statements in Python (if/elif/else)

4.2 Bucles

Cuando queremos hacer algo más de una vez, necesitamos recurrir a un **bucle**. En esta sección veremos las distintas sentencias en Python que nos permiten repetir un bloque de código.[1]

4.2.1 La sentencia `while`

El primer mecanismo que existe en Python para repetir instrucciones es usar la sentencia `while`. La semántica tras esta sentencia es: «Mientras se cumpla la condición haz algo». Veamos un sencillo bucle que muestra por pantalla los números del 1 al 4:

```
>>> value = 1

>>> while value <= 4:
...     print(value)
...     value += 1
...
1
2
3
4
```

[1] Foto original de portada por Gary Lopater en Unsplash.

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/RgM2HYn

La condición del bucle se comprueba en cada nueva repetición. En este caso chequeamos que la variable `value` sea menor o igual que 4. Dentro del cuerpo del bucle estamos incrementando esa variable en 1 unidad.

Romper un bucle while

Python ofrece la posibilidad de *romper* o finalizar un bucle *antes de que se cumpla la condición de parada*. Supongamos un ejemplo en el que estamos buscando el primer número múltiplo de 3 yendo desde 20 hasta 1:

```
>>> num = 20

>>> while num >= 1:
...     if num % 3 == 0:
...         print(num)
...         break
...     num -= 1
...
18
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/wfrKnHl

Como hemos visto en este ejemplo, `break` nos permite finalizar el bucle una vez que hemos encontrado nuestro objetivo: el primer múltiplo de 3. Pero si no lo hubiéramos encontrado, el bucle habría seguido decrementando la variable `num` hasta valer 0, y la condición del bucle `while` hubiera resultado falsa.

Comprobar la rotura

Nivel intermedio

Python nos ofrece la posibilidad de **detectar si el bucle ha acabado de forma ordinaria**, esto es, ha finalizado por no cumplirse la condición establecida. Para ello podemos hacer uso de la sentencia `else` como parte del propio bucle. Si el bucle while finaliza normalmente (sin llamada a `break`) el flujo de control pasa a la sentencia opcional `else`.

Veamos un ejemplo en el que tratamos de encontrar un múltiplo de 9 en el rango $[1, 8]$ (es obvio que no sucederá):

```
>>> num = 8

>>> while num >= 1:
...     if num % 9 == 0:
...         print(f'{num} is a multiple of 9!')
...         break
...     num -= 1
... else:
...     print('No multiples of 9 found!')
...
No multiples of 9 found!
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/CgYQFiA

Continuar un bucle

Nivel intermedio

Hay situaciones en las que, en vez de romper un bucle, nos interesa **saltar adelante hacia la siguiente repetición**. Para ello Python nos ofrece la sentencia `continue` que hace precisamente eso, descartar el resto del código del bucle y saltar a la siguiente iteración.

Veamos un ejemplo en el que usaremos esta estrategia para mostrar todos los números en el rango $[1, 20]$ ignorando aquellos que sean múltiplos de 3:

```
>>> num = 21

>>> while num >= 1:
...     num -= 1
...     if num % 3 == 0:
...         continue
...     print(num, end=', ')  # Evitar salto de línea
...
20, 19, 17, 16, 14, 13, 11, 10, 8, 7, 5, 4, 2, 1,
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/YgYQ3m6

Bucle infinito

Si no establecemos correctamente la **condición de parada** o bien el valor de alguna variable está fuera de control, es posible que lleguemos a una situación de bucle infinito, del que nunca podamos salir. Veamos un ejemplo de esto:

```
>>> num = 1

>>> while num != 10:
...     num += 2
...
# CTRL-C
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

El problema que surje es que la variable `num` toma los valores `1, 3, 5, 7, 9, 11, ...` por lo que nunca se cumple la condición de parada del bucle. Esto hace que repitamos «eternamente» la instrucción de incremento.

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/AfrZroa

Una posible solución a este error es reescribir la condición de parada en el bucle:

```
>>> num = 1

>>> while num < 10:
...     num += 2
...
```

Truco: Para abortar una situación de *bucle infinito* podemos pulsar en el teclado la combinación `CTRL-C`. Se puede ver reflejado en el intérprete de Python por `KeyboardInterrupt`.

Ejercicio

Escriba un programa que calcule la distancia hamming entre dos *cadenas de texto* de la misma longitud.

Ejemplo

- Entrada: `0001010011101` y `0000110010001`
- Salida: `4`

4.2.2 La sentencia `for`

Python permite recorrer aquellos tipos de datos que sean **iterables**, es decir, que admitan *iterar*[2] sobre ellos. Algunos ejemplos de tipos y estructuras de datos que permiten ser iteradas (*recorridas*) son: cadenas de texto, listas, diccionarios, ficheros, etc. La sentencia `for` nos permite realizar esta acción.

A continuación se plantea un ejemplo en el que vamos a recorrer (iterar) una cadena de texto:

```
>>> word = 'Python'

>>> for letter in word:
...     print(letter)
...
P
y
t
h
o
n
```

La clave aquí está en darse cuenta que el bucle va tomando, en cada iteración, cada uno de los elementos de la variable que especifiquemos. En este caso concreto `letter` va tomando cada una de las letras que existen en `word`, porque una cadena de texto está formada por elementos que son caracteres.

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/Pft6R2e

Importante: La variable que utilizamos en el bucle `for` para ir tomando los valores puede tener **cualquier nombre**. Al fin y al cabo es una variable que definimos según nuestras necesidades. Tener en cuenta que se suele usar un nombre en singular.

Romper un bucle for

Una sentencia break dentro de un `for` rompe el bucle, *igual que veíamos* para los bucles `while`. Veamos un ejemplo con el código anterior. En este caso vamos a recorrer una cadena de texto y pararemos el bucle cuando encontremos una letra *t* minúscula:

```
>>> word = 'Python'

```

(continué en la próxima página)

[2] Realizar cierta acción varias veces. En este caso la acción es tomar cada elemento.

(proviene de la página anterior)

```
>>> for letter in word:
...     if letter == 't':
...         break
...     print(letter)
...
P
y
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/zfyqkbJ

Truco: Tanto la *comprobación de rotura de un bucle* como la *continuación a la siguiente iteración* se llevan a cabo del mismo modo que hemos visto con los bucles de tipo `while`.

Ejercicio

Dada una cadena de texto, indique el número de vocales que tiene.

Ejemplo

- Entrada: `Supercalifragilisticoespialidoso`
- Salida: `15`

Secuencias de números

Es muy habitual hacer uso de secuencias de números en bucles. Python no tiene una instrucción específica para ello. Lo que sí aporta es una función `range()` que devuelve un *flujo de números* en el rango especificado. Una de las grandes ventajas es que la «lista» generada no se construye explícitamente, sino que cada valor se genera bajo demanda. Esta técnica mejora el consumo de recursos, especialmente en términos de memoria.

La técnica para la generación de secuencias de números es muy similar a la utilizada en los *«slices»* de cadenas de texto. En este caso disponemos de la función `range(start, stop, step)`:

- **start**: Es *opcional* y tiene valor por defecto **0**.
- **stop**: es *obligatorio* (siempre se llega a 1 menos que este valor).
- **step**: es *opcional* y tiene valor por defecto **1**.

`range()` devuelve un *objeto iterable*, así que iremos obteniendo los valores paso a paso con una sentencia `for ... in`[3]. Veamos diferentes ejemplos de uso:

Rango: $[0, 1, 2]$

```
>>> for i in range(0, 3):
...     print(i)
...
0
1
2

>>> for i in range(3):
...     print(i)
...
0
1
2
```

Rango: $[1, 3, 5]$

```
>>> for i in range(1, 6, 2):
...     print(i)
...
1
3
5
```

Rango: $[2, 1, 0]$

```
>>> for i in range(2, -1, -1):
...     print(i)
...
2
1
0
```

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/vfywE45

Truco: Se suelen utilizar nombres de variables `i`, `j`, `k` para lo que se denominan **contadores**. Este tipo de variables toman valores numéricos enteros como en los ejemplos anteriores. No conviene generalizar el uso de estas variables a situaciones en las que, claramente, tenemos la posibilidad de asignar un nombre semánticamente más significativo. Esto viene de tiempos

[3] O convertir el objeto a una secuencia como una lista.

antiguos en FORTRAN donde `i` era la primera letra que tenía valor entero por defecto.

Ejercicio

Determine si un número dado es un número primo.

No es necesario implementar ningún algoritmo en concreto. La idea es probar los números menores al dado e ir viendo si las divisiones tienen resto cero o no.

¿Podrías optimizar tu código? ¿Realmente es necesario probar con tantos divisores?

Ejemplo

- Entrada: `11`
- Salida: `Es primo`

Usando el guión bajo

Nivel avanzado

Hay situaciones en las que **no necesitamos usar la variable** que toma valores en el rango, sino que únicamente queremos repetir una acción un número determinado de veces.

Para estos casos se suele recomendar usar el **guión bajo _** como **nombre de variable**, que da a entender que no estamos usando esta variable de forma explícita:

```
>>> for _ in range(10):
...     print('Repeat me 10 times!')
...
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
Repeat me 10 times!
```

4.2.3 Bucles anidados

Como ya vimos en las *sentencias condicionales*, el *anidamiento* es una técnica por la que incluimos distintos niveles de encapsulamiento de sentencias, unas dentro de otras, con mayor nivel de profundidad. En el caso de los bucles también es posible hacer anidamiento.

Veamos un ejemplo de 2 bucles anidados en el que generamos todas las tablas de multiplicar:

```
>>> for i in range(1, 10):
...     for j in range(1, 10):
...         result = i * j
...         print(f'{i} * {j} = {result}')
...
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
1 x 4 = 4
1 x 5 = 5
1 x 6 = 6
1 x 7 = 7
1 x 8 = 8
1 x 9 = 9
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
3 x 6 = 18
3 x 7 = 21
3 x 8 = 24
3 x 9 = 27
4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
4 x 5 = 20
4 x 6 = 24
4 x 7 = 28
```

(continué en la próxima página)

(proviene de la página anterior)

```
4 x 8 = 32
4 x 9 = 36
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
6 x 1 = 6
6 x 2 = 12
6 x 3 = 18
6 x 4 = 24
6 x 5 = 30
6 x 6 = 36
6 x 7 = 42
6 x 8 = 48
6 x 9 = 54
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
7 x 6 = 42
7 x 7 = 49
7 x 8 = 56
7 x 9 = 63
8 x 1 = 8
8 x 2 = 16
8 x 3 = 24
8 x 4 = 32
8 x 5 = 40
8 x 6 = 48
8 x 7 = 56
8 x 8 = 64
8 x 9 = 72
9 x 1 = 9
9 x 2 = 18
9 x 3 = 27
9 x 4 = 36
9 x 5 = 45
9 x 6 = 54
9 x 7 = 63
```

(continué en la próxima página)

(proviene de la página anterior)

```
9 x 8 = 72
9 x 9 = 81
```

Lo que está ocurriendo en este código es que, para cada valor que toma la variable `i`, la otra variable `j` toma todos sus valores. Como resultado tenemos una combinación completa de los valores en el rango especificado.

Ejecución **paso a paso** a través de *Python Tutor*:

https://cutt.ly/vfyeWvj

Nota:

- Podemos añadir todos los niveles de anidamiento que queramos. Eso sí, hay que tener en cuenta que cada nuevo nivel de anidamiento supone un importante aumento de la complejidad ciclomática de nuestro código, lo que se traduce en mayores tiempos de ejecución.
- Los bucles anidados también se pueden aplicar en la sentencia `while`.

Ejercicio

Imprima los 100 primeros números de la sucesión de Fibonacci: $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots$

EJERCICIOS DE REPASO

1. Escriba un programa en Python que realice las siguientes 9 multiplicaciones. ¿Nota algo raro? (`solución`)

$$1 \cdot 1$$
$$11 \cdot 11$$
$$111 \cdot 111$$
$$\vdots$$
$$111111111 \cdot 111111111$$

2. Escriba un programa en Python que acepte una cadena de texto e indique si todos sus caracteres son alfabéticos (`solución`).

   Entrada: hello-world

   Salida: Se han encontrado caracteres no alfabéticos

3. Escriba un programa en Python que acepte un número entero n y realice el siguiente cálculo de productos sucesivos (`solución`):

$$\prod_{i=1}^{n} x_i^2 = x_0^2 \cdot x_1^2 \cdot x_2^2 \cdot \cdots \cdot x_n^2$$

4. Escriba un programa en Python que acepte dos cadenas de texto y compute el producto cartesiano letra a letra entre ellas (`solución`).

   Entrada: cadena1=abc; cadena2=123
   Salida: a1 a2 a3 b1 b2 b3 c1 c2 c3

5. Escriba un programa en Python que acepte dos valores enteros (x e y) que representarán un punto (objetivo) en el plano. El programa simulará el movimiento de un «caballo» de ajedrez moviéndose de forma alterna: 2 posiciones en $x + 1$ posición en y. El siguiente movimiento que toque sería para moverse 1 posición en $x + 2$ posiciones en y. El programa deberá ir mostrando los puntos por los que va pasando el «caballo» hasta llegar al punto objetivo (`solución`).

   Entrada: objetivo_x=7; objetivo_y=8;
   Salida: (0, 0) (1, 2) (3, 3) (4, 5) (6, 6) (7, 8)

AMPLIAR CONOCIMIENTOS

- The Python range() Function
- How to Write Pythonic Loops
- For Loops in Python (Definite Iteration)
- Python «while» Loops (Indefinite Iteration)