Control de Flujo

Vamos con la corriente

Operadores aritméticos

Los operadores aritméticos se utilizan para realizar operaciones aritméticas, es decir, para manipular datos numéricos a través de operaciones matemáticas tales como la adición, sustracción o multiplicación...

```
a = 10
b = 2

a + b
```

```
12
```

```
a * b
```

```
20
```

+, *, bla bla bla

Operadores de asignación (Trucos de Python)

Los operadores de asignación son aquellos que permiten dar un valor a una variable o modificarlo. Python tiene ocho diferentes operadores de asignación: un operador de asignación simple y siete operadores de asignación compuestos.

- = Asignación simple a=b
- += Asignación de adición a+=b Equivalente simple a=a+b
- -= Asignación de sustracción a-=b Equivalente simple a=a-b
- *= Asignación de multiplicación a*=b Equivalente simple a=a*b
- /= Asignación de división a/=b Equivalente simple a=a/b
- %= Asignación de módulo a%=b Equivalente simple a=a%b
- //= Asignación de división entera a//=b Equivalente simple a=a//b

```
a = a + 1
a
```

```
11
```

```
a += 1
```

```
a
```

```
12
```

El operador de asignación simple es el símbolo igual (=) y las operaciones realizadas sobre él siempre tienen la sintaxis: `variable = expresión`. En este tipo de operación, primero, se resuelve la expresión de la derecha y el valor resultante se asigna a la variable de la izquierda.

```
# Coge un poco de tiempo, y práctica
```

Operadores relacionales

Los operadores relacionales son símbolos utilizados para comparar dos valores o expresiones. El resultado de la evaluación con estos operadores puede ser True, si la comparación es verdadera, o False, si la comparación es falsa.

- `==` Igual a `a==b`
- `!=` Diferente de `a!=b`
- `>` Mayor que `a>b`
- `<` Menor que `a<b`
- `>=` Mayor o igual que `a>=b`
- `<=` Menor o igual que `a<=b` Nota la diferencia entre un signo igual (=), que es una asignación, y un doble signo igual (==), que es un operador relacional

**Consejo: Utiliza paréntesis para mantener el orden de las operaciones.

```
a = 10
```

```
a == 10
```

```
True
```

```
a != 10
```

```
False
```

```
a > b
```

```
True
```

```
a < b
```

```
False
```

```
a <= b
```

```
False
```

```
b <= a
```

```
True
```

Con solo estos dos valores `True`|`False` podemos crear toda una rama de las matemáticas llamada Álgebra Booleana. Mientras que en el Álgebra regular las operaciones básicas son la adición y la multiplicación, las operaciones principales en el Álgebra Booleana son la conjunción (y), la disyunción (o), y la negación (no). **Es el formalismo utilizado para describir las operaciones lógicas**.

En Python escribimos estas operaciones como:

- `==`
- `x` y `y`
- `x` o `y`
- no `x`

También podemos incluir el OR EXCLUSIVO, que es verdadero cuando uno y solo uno de los operandos es verdadero, pero estrictamente debes saber que se deriva de los tres básicos. Su representación es ^, el sombrero o caret.

- Extra: `^` operador caret

Aunque el significado de estas operaciones es claro, podemos definirlas completamente con la llamada "tabla de verdad":

```
>>>
x       y       x or y
---------------------
False   False   False
False   True    True
True    False   True
True    True    True

x       y       x and y
----------------------
False   False   False
False   True    False
True    False   False
True    True    True

x       not x
-------------
False   True
True    False

x       y       x ^ y
---------------------
False   False   False
False   True    True
True    False   True
True    True    False
>>>
```

- **igualdad** vs **identidad**

Algunos documentos

`==` -> apunta al valor `is` -> busca la identidad

```
# Ejemplo 1: Comparando números
a = 5
b = 5

print(a == b) # True, porque los valores de 'a' y 'b' son iguales
```

```
True
```

```
print(a is b) # True, porque 'a' y 'b' hacen referencia al mismo
objeto en memoria (optimización de Python para números pequeños)
```

```
True
```

```
# Ejemplo 2: Comparando listas
lista1 = [1, 2, 3]
lista2 = [1, 2, 3]

print(lista1 == lista2) # True, porque los valores de 'lista1' y
'lista2' son iguales
```

```
True
```

```
print(lista1 is lista2) # False, porque 'lista1' y 'lista2' hacen
referencia a diferentes objetos en memoria
```

```
False
```

- El operador **no**. El operador **no** es un operador que devuelve el valor opuesto de la expresión evaluada. Si la expresión tiene el valor True, devuelve False. Por el contrario, si la expresión tiene el valor False, devuelve True.

```
not(lista1 == lista2)
```

```
False
```

- El operador **y** evalúa si ambas expresiones son verdaderas. Si ambas expresiones son verdaderas, devuelve True. Si alguna de las expresiones es falsa, devuelve False. Estos tipos de tablas se conocen formalmente como "tablas de verdad".

```
# Same to (lista1 == lista2) & (a == lista2)
(lista1 == lista2) and (a == lista2)
```

```
False
```

- El operador **o** evalúa si alguna de las expresiones es verdadera, es decir, devuelve True si alguna de las expresiones es verdadera y False cuando ambas expresiones son falsas.

```
# Same to (lista1 == lista2) | (a == lista2)
(lista1 == lista2) or (a == lista2)
```

```
True
```

Ejercicio: Explorando los Operadores Relacionales

En este ejercicio, exploraremos los diversos operadores relacionales disponibles en Python. Estos operadores se utilizan principalmente para establecer relaciones entre diferentes valores y son esenciales en estructuras de toma de decisiones, aunque no los usaremos en esa capacidad en este ejercicio. Utilizando estos operadores, por favor realiza las siguientes tareas:

Tarea 1: Operador Igual a (==)

1. Crea dos cadenas y verifica si son iguales usando `==`. Imprime el resultado.
2. Crea dos listas con elementos iguales y verifica si son iguales usando `==`. Imprime el resultado.

Tarea 2: Operador No Igual a (!=)

1. Crea dos enteros y verifica si no son iguales usando `!=`. Imprime el resultado.
2. Usa la función `input()` para obtener dos entradas del usuario y verifica si no son iguales usando `!=`. Imprime el resultado.

Tarea 3: Operadores Mayor que y Menor que (>, <)

1. Crea dos números de punto flotante y verifica si uno es mayor que el otro usando `>`. Imprime el resultado.
2. Crea dos números de punto flotante y verifica si uno es menor que el otro usando `<`. Imprime el resultado.

Tarea 4: Operadores Mayor o Igual a y Menor o Igual a (>=, <=)

1. Crea dos enteros y verifica si uno es mayor o igual al otro usando `>=`. Imprime el resultado.
2. Crea dos enteros y verifica si uno es menor o igual al otro usando `<=`. Imprime el resultado.

```
# tu codígo aquí

# 3.1.1. Tarea 1: Operador Igual a (==) y Comprobación de la Posición de Memoria (is)

# Verificando si dos cadenas son iguales usando == y si ocupan la misma posición de memoria usando is
cadena1 = "Hola"
cadena2 = "Hola"
resultado_cadenas = cadena1 == cadena2
misma_posicion_cadenas = cadena1 is cadena2
print("¿Las cadenas son iguales?:", resultado_cadenas)
print("¿Las cadenas ocupan la misma posición en memoria?:", misma_posicion_cadenas)

¿Las cadenas son iguales?: True
¿Las cadenas ocupan la misma posición en memoria?: True
```

```
# Verificando si dos listas con elementos iguales son iguales usando
== y si ocupan la misma posición de memoria usando is
lista1 = [1, 2, 3]
lista2 = [1, 2, 3]
resultado_listas = lista1 == lista2
misma_posicion_listas = lista1 is lista2
print("¿Las listas son iguales?:", resultado_listas)
print("¿Las listas ocupan la misma posición en memoria?:",
misma_posicion_listas)
```

```
¿Las listas son iguales?: True
¿Las listas ocupan la misma posición en memoria?: False
```

```
# 3.1.2. Tarea 2: Operador No Igual a (!=)

# Verificando si dos enteros no son iguales usando != y si ocupan la
misma posición de memoria usando is
entero1 = 5
entero2 = 10
resultado_enteros = entero1 != entero2
misma_posicion_enteros = entero1 is entero2
print("¿Los enteros son diferentes?:", resultado_enteros)
print("¿Los enteros ocupan la misma posición en memoria?:",
misma_posicion_enteros)
```

```
¿Los enteros son diferentes?: True
¿Los enteros ocupan la misma posición en memoria?: False
```

```
# Usando la función input() para obtener dos entradas del usuario y
verificar si no son iguales usando != y si ocupan la misma posición de
memoria usando is
entrada1 = input("Introduce la primera entrada: ")
entrada2 = input("Introduce la segunda entrada: ")
resultado_entradas = entrada1 != entrada2
misma_posicion_entradas = entrada1 is entrada2
print("¿Las entradas son diferentes?:", resultado_entradas)
print("¿Las entradas ocupan la misma posición en memoria?:",
misma_posicion_entradas)
```

```
Introduce la primera entrada:  a
Introduce la segunda entrada:  a
```

```
¿Las entradas son diferentes?: False
¿Las entradas ocupan la misma posición en memoria?: True
```

```
# 3.1.3. Tarea 3: Operadores Mayor que y Menor que (>, <)

# Verificando si un número de punto flotante es mayor que otro usando
> y si ocupan la misma posición de memoria usando is
numero1 = 3.14
numero2 = 2.71
```

```
resultado_mayor = numero1 > numero2
misma_posicion_numeros = numero1 is numero2
print("¿El primer número es mayor que el segundo?:", resultado_mayor)
print("¿Los números ocupan la misma posición en memoria?:",
misma_posicion_numeros)
```

```
¿El primer número es mayor que el segundo?: True
¿Los números ocupan la misma posición en memoria?: False
```

```
# Verificando si un número de punto flotante es menor que otro usando
< y si ocupan la misma posición de memoria usando is
numero3 = 1.23
numero4 = 4.56
resultado_menor = numero3 < numero4
misma_posicion_numeros2 = numero3 is numero4
print("¿El primer número es menor que el segundo?:", resultado_menor)
print("¿Los números ocupan la misma posición en memoria?:",
misma_posicion_numeros2)
```

```
¿El primer número es menor que el segundo?: True
¿Los números ocupan la misma posición en memoria?: False
```

```
# 3.1.4. Tarea 4: Operadores Mayor o Igual a y Menor o Igual a (>=,
<=)

# Verificando si un entero es mayor o igual que otro usando >= y si
ocupan la misma posición de memoria usando is
entero3 = 7
entero4 = 7
resultado_mayor_igual = entero3 >= entero4
misma_posicion_enteros2 = entero3 is entero4
print("¿El primer entero es mayor o igual que el segundo?:",
resultado_mayor_igual)
print("¿Los enteros ocupan la misma posición en memoria?:",
misma_posicion_enteros2)
```

```
¿El primer entero es mayor o igual que el segundo?: True
¿Los enteros ocupan la misma posición en memoria?: True
```

```
# Verificando si un entero es menor o igual que otro usando <= y si
ocupan la misma posición de memoria usando is
entero5 = 9
entero6 = 12
resultado_menor_igual = entero5 <= entero6
misma_posicion_enteros3 = entero5 is entero6
print("¿El primer entero es menor o igual que el segundo?:",
resultado_menor_igual)
print("¿Los enteros ocupan la misma posición en memoria?:",
misma_posicion_enteros3)
```

```
¿El primer entero es menor o igual que el segundo?: True
¿Los enteros ocupan la misma posición en memoria?: False
```

Bucles y esas cosas

En programación, un **bucle** es un concepto fundamental que permite repetir un bloque de código varias veces. Facilita la ejecución de tareas repetitivas sin necesidad de escribir las mismas líneas de código una y otra vez, promoviendo tanto la eficiencia como la legibilidad. La idea básica es recorrer un bloque de código mientras una condición especificada permanezca verdadera.

Generalmente, hay dos tipos de bucles que encontrarás:

1. **Bucle For**: Generalmente se usa para iterar sobre una colección de elementos, como elementos en una lista, tupla o cadena. Ejecutará un bloque de código para cada elemento en la colección, lo que hace más simple y limpio realizar la misma acción en múltiples elementos.

2. **Bucle While**: Este tipo de bucle continúa ejecutando un bloque de código mientras una condición particular se cumpla. Es crucial actualizar las variables que influyen en la condición dentro del bucle, para evitar crear un bucle infinito.

Los bucles son herramientas increíblemente poderosas en programación, ayudando a ahorrar tiempo y reducir el potencial de errores en tu código. A medida que avances en tu aprendizaje, encontrarás que los bucles son componentes esenciales en muchas soluciones de programación, facilitando la creación de soluciones complejas, eficientes e innovadoras para una amplia gama de problemas.

En las siguientes secciones, exploraremos ambos tipos de bucles con más detalle, con ejemplos para ilustrar su uso y versatilidad en tareas de programación.

¡Feliz bucleo!

Pseudocódigo

Antes de sumergirnos en el mundo de los bucles, tomemos un momento para entender qué es el "Pseudocódigo". El pseudocódigo es una descripción de alto nivel del funcionamiento de un programa de computadora o algoritmo. Utiliza las convenciones estructurales de los lenguajes de programación, pero está diseñado para ser leído por humanos en lugar de máquinas. Esto lo convierte en una herramienta poderosa en la fase de planificación y diseño de un proyecto.

¿Por qué es útil el pseudocódigo?

El uso de pseudocódigo ofrece múltiples ventajas:

- **Claridad Conceptual**: Permite a los programadores visualizar la lógica del algoritmo sin distraerse con la sintaxis de un lenguaje específico. Esto es especialmente útil cuando se trabaja en equipo, ya que todos pueden comprender la lógica sin necesidad de conocer el mismo lenguaje de programación.

- **Iteración Rápida**: Al estar libre de la sintaxis rigurosa, los desarrolladores pueden modificar y ajustar rápidamente la lógica del algoritmo. Esto ayuda a identificar problemas en las fases iniciales y a experimentar con diferentes enfoques antes de la implementación.

- **Documentación Eficiente**: El pseudocódigo puede servir como una forma de documentación técnica que complementa el código real. Puede ser útil para que otros desarrolladores entiendan el propósito y la estructura del programa.

Estructura del Pseudocódigo

En el pseudocódigo, no nos preocupamos por la sintaxis precisa que un lenguaje de programación pueda requerir. En cambio, nos enfocamos en expresar la lógica y los pasos necesarios para resolver un problema o lograr una tarea, a menudo en lenguaje natural mezclado con algunas estructuras de programación comunes. Esto significa que el pseudocódigo no tiene una sintaxis estricta y puede variar ampliamente.

Objetivos del Pseudocódigo

El objetivo principal de escribir pseudocódigo es delinear la estructura del programa y su flujo lógico sin quedar atrapados en los detalles específicos de un lenguaje de programación particular. Esto lo convierte en una herramienta útil para planificar y diseñar programas, donde el énfasis está en qué hacer en lugar de cómo hacerlo.

Además, el pseudocódigo ayuda a:

- **Mejorar la Comunicación**: Facilita la discusión de algoritmos entre desarrolladores y otros interesados, independientemente del lenguaje que se utilice.
- **Identificar Errores Lógicos**: Permite a los desarrolladores detectar problemas en la lógica antes de escribir código, ahorrando tiempo y esfuerzo en la depuración posterior.
- **Optimizar el Diseño**: Ayuda a refinar el algoritmo, asegurando que esté bien estructurado antes de la implementación.

Características Generales del Pseudocódigo

1. **Simplicidad**: Debe ser simple y fácil de entender, evitando jergas técnicas innecesarias.
2. **Claridad**: Cada paso está claramente definido sin ambigüedad, facilitando la lectura y comprensión.
3. **Independencia del Lenguaje**: No se adhiere a la sintaxis de ningún lenguaje de programación específico, lo que permite flexibilidad en su uso.
4. **Enfoque en la Lógica**: El enfoque principal está en la lógica y flujo del algoritmo, más que en la sintaxis específica de un lenguaje de programación.
5. **Flexibilidad**: Puede adaptarse según las necesidades del programador o del equipo, permitiendo diferentes estilos de escritura.
6. **Uso de Comentarios**: Se pueden utilizar comentarios para explicar secciones del pseudocódigo, brindando contexto adicional y ayudando a otros a entender la lógica.

Conclusión

A medida que comiences a trabajar con bucles y otras estructuras de programación, encontrarás que crear pseudocódigo puede ser un valioso primer paso en el proceso de codificación. Esto te ayudará a organizar tus pensamientos y crear un mapa de ruta para tu programa antes de comenzar a escribir código real. La práctica de escribir pseudocódigo no solo facilitará la codificación, sino que también mejorará tu capacidad para resolver problemas de manera efectiva y te convertirá en un programador más versátil y competente.

```
# 1. Generar la Sucesión de Fibonacci
# OBJETIVO: Generar la sucesión de Fibonacci hasta el n-ésimo término.
```

```
# NECESITO: Un número entero n.
# 1. Inicializar fib_sequence como lista vacía
# 2. a = 0, b = 1
# 3. Para i desde 0 hasta n - 1:
    # a. Añadir a fib_sequence el valor de a
    # b. Actualizar a = b
    # c. Actualizar b = a + b
# 4. Imprimir fib_sequence
# DEVUELVO: Una lista con los primeros n términos de Fibonacci.


# 2. Generar un Buscador de Números Múltiplos de Otro Número
# OBJETIVO: Encontrar todos los múltiplos de un número hasta un límite
dado.
# NECESITO: Un número (numero) y un límite (limite).
# 1. Inicializar multiplos como lista vacía
# 2. Para i desde 1 hasta limite:
    # a. Si i es múltiplo de numero:
        # i. Añadir i a multiplos
# 3. Imprimir multiplos
# DEVUELVO: Una lista de múltiplos de numero.


# 3. Generar un Buscador de Números Primos
# OBJETIVO: Encontrar todos los números primos hasta un límite dado.
# NECESITO: Un límite (limite).
# 1. Inicializar primos como lista vacía
# 2. Para i desde 2 hasta limite:
    # a. Si i es primo:
        # i. Añadir i a primos
# 3. Imprimir primos
# DEVUELVO: Una lista de números primos.


# 4. Generar un Método de Newton-Raphson
# OBJETIVO: Encontrar una raíz de una función usando el método de
Newton-Raphson.
# NECESITO: Una función (func), su derivada (derivada), un valor
inicial (x0), una tolerancia y un número máximo de iteraciones.
# 1. Asignar x = x0
# 2. Para i desde 0 hasta max_iter:
    # a. Calcular x_new usando el método
    # b. Si |x_new - x| < tolerancia:
        # i. Imprimir x_new
        # ii. Terminar
    # c. Actualizar x = x_new
# 3. Imprimir None (si no converge)
# DEVUELVO: Un valor que es una aproximación a la raíz, o None si no
converge.
```

Tarjeta de información rápida:

En Python, la función `range()` se utiliza para generar una secuencia de números con el tiempo. Se utiliza ampliamente en bucles para controlar el número de iteraciones. La función tiene tres parámetros: start, stop y step.

- **Start (Inicio)**: Este parámetro especifica el punto de inicio de la secuencia. Es opcional, y el valor por defecto es 0.
- **Stop (Detener)**: Este parámetro especifica el punto final de la secuencia. Este valor no está incluido en la secuencia.
- **Step (Paso)**: Este parámetro especifica el incremento entre cada número en la secuencia. Es opcional, y el valor por defecto es 1.

Aquí está la sintaxis para la función `range()`:

- range(start, stop, step)

Algunos ejemplos:

```
for i in range(1, 6):
    print(i)

1
2
3
4
5

# práctica aquí
```

Más complejo:

```
# Esta es la lista original de listas donde cada elemento es una lista que contiene enteros.
original = [[1, 2], [3], [5, 6, 9]]

# Se inicializa una nueva lista vacía para almacenar los elementos individuales después de aplanar la lista original.
new_list = []

# El bucle exterior itera a través de cada lista individual dentro de la lista original.
for individual_list in original:
    # El bucle interior itera a través de cada elemento en la lista individual actualmente seleccionada (del bucle exterior).
    for each_element in individual_list:
        # El elemento actual de la lista individual se añade a la nueva lista.
        new_list.append(each_element)
```

```
# Imprimiendo la new_list para ver el resultado final.
print(new_list)
```

```
[1, 2, 3, 5, 6, 9]
```

Debug de lo que está pasando:

```
# Estamos definiendo una lista llamada 'bigger_list' que contiene
varias otras listas, una de las cuales está anidada
bigger_list = [[1, 2], [3], [5, 6, 9, [1, 2]]]

# Estamos iniciando un bucle for para iterar sobre cada 'small_list'
presente dentro de la 'bigger_list'
for small_list in bigger_list:
    # Dentro del bucle, estamos imprimiendo la 'small_list' actual
sobre la que el bucle está iterando
    print(small_list)
    # Una vez que se alcanza el final del bucle, vuelve al inicio para
procesar la siguiente 'small_list' en la 'bigger_list'
```

```
[1, 2]
[3]
[5, 6, 9, [1, 2]]
```

Ejercicio:

Trabajas en el departamento de ventas de una empresa. Al final de cada semana, necesitas generar un informe de las ventas totales realizadas cada día. Tienes una lista donde cada elemento representa las ventas realizadas en cada día de la semana (de lunes a domingo). Tu tarea es calcular las ventas totales de la semana.

Para completar este ejercicio, necesitas seguir los pasos dados:

1. **Inicializar Variables**: Antes de comenzar el bucle, inicializa una variable `total_sales` a 0. Esto se utilizará para acumular las ventas de cada día. Adicionalmente, inicializa `max_sales` a 0.

2. **Creando un Bucle**: Crea un bucle for donde iteres sobre cada elemento en la lista `sales_data`. Puedes usar una estructura de bucle como esta: for day in sales_data:

3. **Calculando las Ventas Totales**: Dentro del bucle, suma las ventas del día actual a `total_sales`. Puedes hacer esto usando una declaración de asignación como esta: total_sales += day

4. **Imprimiendo los Resultados**: Después del bucle, imprime el `total_sales` junto con mensajes apropiados para mostrar las ventas totales de la semana y el día con máximas ventas.

```
# Lista que representa los datos de ventas de siete días (en unidades)
sales_data = [200, 300, 400, 500, 600, 700, 800]

# Inicializar total_sales a 0 antes de empezar el bucle
total_sales = 0

# Inicializar max_sales antes de empezar el bucle
max_sales = 0

# {YOUR CODE HERE}

# Inicializar un contador para los días
day_counter = 1

# Crear un bucle para iterar sobre cada elemento en sales_data
for day in sales_data:
    # Sumar las ventas del día actual a total_sales
    total_sales += day

    # Comprobar si las ventas del día actual son mayores que max_sales
    if day > max_sales:
        max_sales = day
        max_sales_day = day_counter  # Guardar el día con las máximas
ventas

    # Incrementar el contador de días
    day_counter += 1

# Imprimir las ventas totales de la semana
print(f"Ventas totales de la semana: {total_sales} unidades")

# Imprimir el día con las máximas ventas
print(f"El día con las máximas ventas es el día {max_sales_day} con
{max_sales} unidades")
```

```
Ventas totales de la semana: 3500 unidades
El día con las máximas ventas es el día 7 con 800 unidades
```

La sentencia "If"

Esta es la opción más popular para controlar el flujo de un programa. Las condiciones te permiten elegir entre diferentes caminos dependiendo del valor de una expresión. Cuando quieres ejecutar una acción solo cuando alguna condición es `True`, se utiliza una sentencia "if":

Cosas de sintaxis a considerar:

- indentación: `tab / 5 espacios`
- dos puntos: `:`

```
age = 20
```

```
if age >= 21:
    print("Adult")
```

Como podemos ver, el bloque de código asociado con la condición comienza después del dos puntos ":", con una indentación que determina el bloque de código. Todas las declaraciones pertenecientes al mismo bloque deben tener la misma indentación. El bloque termina cuando la indentación vuelve a la posición inicial de la declaración if. Recuerda que Python utiliza la indentación para identificar bloques de código.

elif

A veces hay más de dos posibilidades. Piensa en algo como: "Si Bob puede entrar a la discoteca, déjalo entrar. Si no, si tiene más de 16 años, recomiéndale la discoteca ligera cercana. Si no, envíalo a casa."

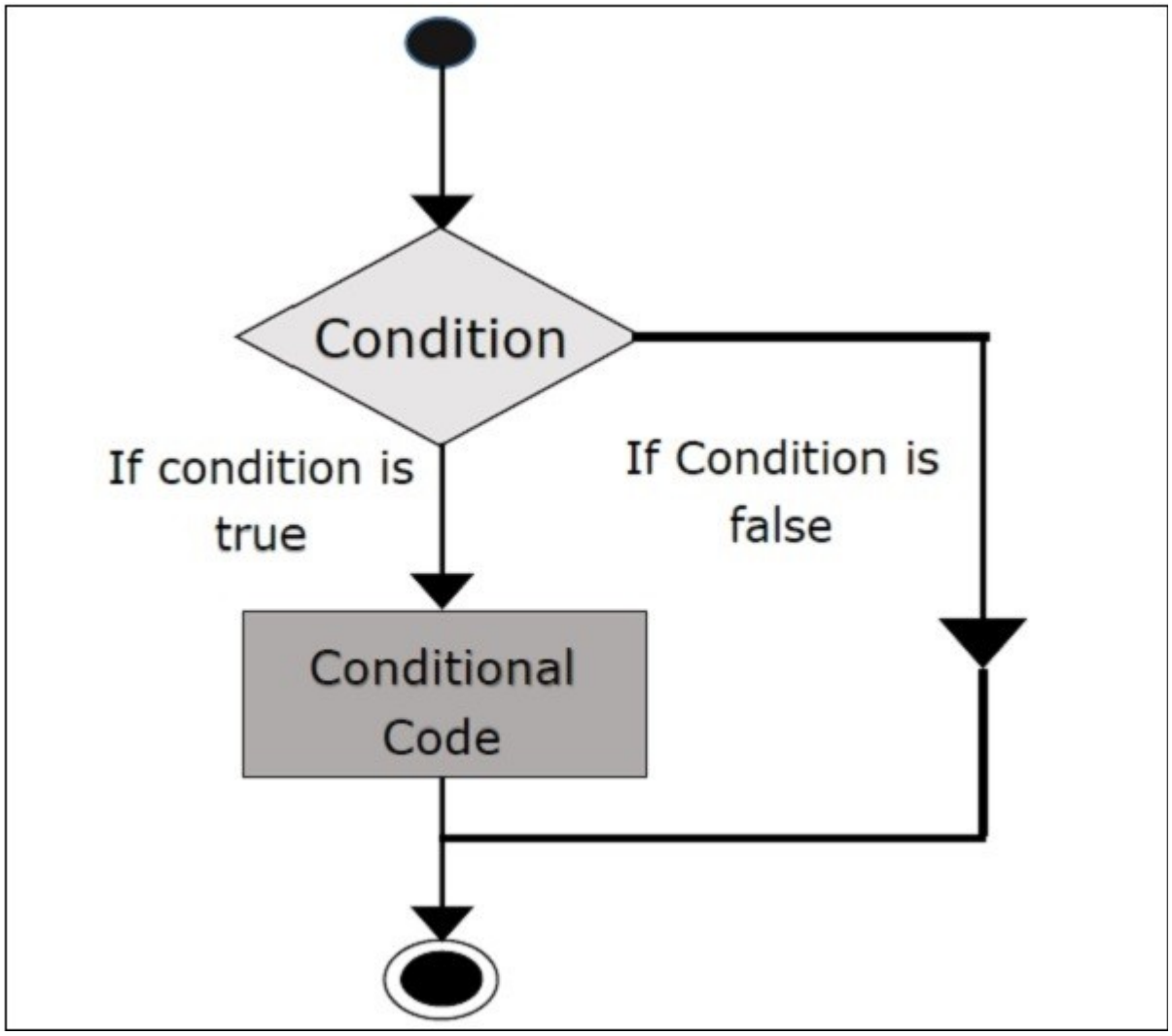
Podemos lograr esto con la cláusula `elif` y los condicionales encadenados:

```
age = 20

if age >= 21:
    print("Adult")

elif age < 21:
    print("Not an adult")
```

```
Not an adult
```

else

A veces quieres que si la condición no se cumple (y solo entonces) se ejecute otra acción. Son acciones mutuamente excluyentes. Esto se logra con la declaración `else`.




```
edad = 14

if edad >= 21:
    print("Adulto")

elif edad >= 16 and edad < 21:
    print("No es adulto")

else:
    print("no condición 1, no condición 2")
```

```
no condición 1, no condición 2
```

En la programación de Python, una instrucción if anidada es una instrucción if que es el objetivo de otra instrucción if. Las instrucciones if anidadas significan una instrucción if dentro de otra instrucción if. Sí, Python nos permite anidar instrucciones if dentro de instrucciones if. Aquí está la forma general de una instrucción if anidada

```python
# 1. Toma la entrada de un usuario (controlando que sea un entero)
edad = int(input("Por favor, introduce tu edad: "))

# 2. Basado en condiciones: devuelve algo o algo más
if edad > 0:
    if edad < 16:
        print("NO puedes entrar")

    elif edad >= 16 and edad < 18:
        print("Club sin alcohol")


    elif edad >= 18:
        print("Puedes entrar")

else:
    print("Lo siento, información incorrecta: ¿valor negativo?")
```

```
Por favor, introduce tu edad:  32

Puedes entrar
```

Ejercicio: Validación de Contraseña Segura

En un sistema de gestión de usuarios, es crucial que las contraseñas sean seguras para proteger la información personal de los usuarios. Tu tarea es crear un script que valide una contraseña según los siguientes criterios:

1. **Asignación de la Contraseña**:
   - Asigna una cadena a la variable `password`. Por ejemplo, puedes usar `"12345678909876dcvbnhjytfgvb+"`.
2. **Verificación de Longitud**:
   - La contraseña debe tener al menos 12 caracteres. Si cumple con este criterio, imprime `"Longitud adecuada"`.
3. **Verificación de Presencia de Letras**:
   - La contraseña debe contener al menos una letra (carácter alfabético). Si es verdadero, imprime `"Contiene letras"`.
4. **Verificación de Presencia de Números**:
   - La contraseña debe contener al menos un número. Si es verdadero, imprime `"Contiene números"`.
5. **Verificación de Presencia de Caracteres Especiales**:
   - La contraseña debe contener al menos un carácter especial (por ejemplo, `!@#$%^&*()`). Si es verdadero, imprime `"Contiene caracteres especiales"`.
6. **Resultado Final**:
   - Si la contraseña cumple con todos los criterios, imprime `"Contraseña segura"`. De lo contrario, imprime `"Contraseña insegura"`.

Escribe el código para cumplir con las instrucciones anteriores y asegúrate de que se impriman las salidas correctas según las verificaciones realizadas.

Hint

La expresión `any(x.isalpha() for x in password)` se utiliza para verificar si hay al menos una letra en la contraseña. Aquí te explico cómo funciona:

- **`any()`**: Esta función toma un iterable (en este caso, una expresión generadora) y devuelve `True` si al menos uno de los elementos del iterable es `True`. Si todos los elementos son `False`, devuelve `False`.
- **`x.isalpha()`**: Este método de cadena devuelve `True` si el carácter `x` es una letra (A-Z, a-z). Si `x` no es una letra, devuelve `False`.
- **`for x in password`**: Este bucle recorre cada carácter en la variable `password`.

Así que la expresión completa revisa cada carácter en la contraseña y devuelve `True` si encuentra al menos una letra, lo que indica que la contraseña cumple con el criterio de contener letras.

```
# Asignar una cadena que representa una contraseña a la variable
'password'
password = "12345678909876dcvbnhjytfgvb+!"

# Inicializar las variables de verificación
length_valid = len(password) >= 12
contains_letter = any(x.isalpha() for x in password)
contains_digit = any(x.isdigit() for x in password)
contains_special = any(x in "!@#$%^&*()" for x in password)

# Verificar longitud
if length_valid:
    print("Longitud adecuada")
else:
    print("Longitud inadecuada")

# Verificar presencia de letras
if contains_letter:
    print("Contiene letras")
else:
    print("No contiene letras")

# Verificar presencia de números
if contains_digit:
    print("Contiene números")
else:
    print("No contiene números")

# Verificar presencia de caracteres especiales
```

```
if contains_special:
    print("Contiene caracteres especiales")
else:
    print("No contiene caracteres especiales")

# Resultado final
if length_valid and contains_letter and contains_digit and
contains_special:
    print("Contraseña segura")
else:
    print("Contraseña insegura")
```

```
Longitud adecuada
Contiene letras
Contiene números
Contiene caracteres especiales
Contraseña segura
```

Quick tip:

```
if no sé qué:
    no sé
elif: # Esta línea debería ser corregida para reflejar una condición
válida, ya que "elif" siempre requiere una condición
    algo más
```

Es computacionalmente más eficiente usar elif, aunque este código funciona, Python entrará y comprobará todos los if

```
# IF & IF: cuando más de una condición puede cumplirse a la vez
    # todos los if se leen
    # todos los if serán leídos

# IF & ELIF: cuando solo una debe cumplirse
    # solo se leen los necesarios
    # mejor: más conservador
```

Extensión de ejercicio:

Utiliza if statements, para calcularl el top de ventas, y que días fueron el máximo.

```
# Lista representando los datos de ventas para siete días (en
unidades)
sales_data = [200, 300, 400, 500, 600, 700, 800]

# Inicializar total_sales a 0 antes de empezar el bucle
total_sales = 0

# Inicializar max_sales antes de empezar el bucle
```

```
max_sales = 0

# Inicializar una lista para rastrear los días con ventas máximas
max_sales_days = []

# {YOUR CODE HERE}

# Crear un bucle para iterar sobre cada elemento en sales_data
for day_index, day_sales in enumerate(sales_data):
    # Sumar las ventas del día actual a total_sales
    total_sales += day_sales

    # Verificar si las ventas del día actual son mayores que max_sales
    if day_sales > max_sales:
        max_sales = day_sales
        # Reiniciar la lista de días con máximas ventas y agregar el
día actual
        max_sales_days = [day_index + 1]
    elif day_sales == max_sales:
        # Si las ventas del día actual son iguales a max_sales,
agregar el día a la lista
        max_sales_days.append(day_index + 1)

# Imprimir las ventas totales de la semana
print(f"Ventas totales de la semana: {total_sales} unidades")

# Imprimir el o los días con las máximas ventas
if len(max_sales_days) == 1:
    print(f"El día con las máximas ventas es el día
{max_sales_days[0]} con {max_sales} unidades")
else:
    print(f"Los días con las máximas ventas son los días {',
'.join(map(str, max_sales_days))} con {max_sales} unidades cada uno")
```

```
Ventas totales de la semana: 3500 unidades
El día con las máximas ventas es el día 7 con 800 unidades
```

Ejercicio: Validación de Entrada al Club

Bob quiere entrar a un club exclusivo que tiene ciertas restricciones de edad. La política del club establece lo siguiente:

- La edad mínima para entrar al club es 16 años.
- La edad máxima para entrar al club es 18 años.
- Si Bob tiene exactamente 18 años, se le permitirá entrar, pero si tiene más de 18 años, no se le permitirá la entrada.

Tu tarea es escribir un programa que imprima si Bob puede entrar al club basado en su edad y la edad mínima permitida.

Requisitos:

1. **Entrada**: Crea dos variables:
   - `edad_bob`: La edad de Bob (un número entero).
   - `edad_minima`: La edad mínima requerida para entrar al club (un número entero).
2. **Verificación de Edad**:
   - Si la edad de Bob es menor que la edad mínima, imprime `"Bob no puede entrar al club"`.
   - Si la edad de Bob está entre 16 y 18 años (inclusive), imprime `"Bob puede entrar al club"`.
   - Si la edad de Bob es mayor que 18 años, imprime `"Bob no puede entrar al club"`.
3. **Bonus**: Asegúrate de que el programa también maneje la situación en que Bob tiene exactamente 16 años y se le permita la entrada.

Escribe el código para cumplir con los requisitos anteriores y asegúrate de que se impriman los mensajes correctos según la edad de Bob.

```
# age
    # club
    # cannot club
    # club: 16-18

# Variables de edad
edad_bob = 17  # Cambia este valor para probar diferentes casos
edad_minima = 16

# Verificación de la entrada al club
if edad_bob < edad_minima:
    print("Bob no puede entrar al club")
elif 16 <= edad_bob <= 18:
    print("Bob puede entrar al club")
else:
    print("Bob no puede entrar al club")

Bob puede entrar al club
```

Ejercicio: Validación de Entrada al Club y Discoteca

Modifica tu programa para que Bob pueda disfrutar de una mejor experiencia según su edad. Las reglas del club son las siguientes:

- La edad mínima para entrar al club es 16 años.
- La edad máxima para entrar al club es 18 años.
- Si Bob tiene exactamente 18 años, se le permitirá entrar.
- Si Bob es mayor de 18 años pero menor o igual a 21, se le indicará que no puede entrar al club.
- Si Bob tiene más de 21 años, se le enviará a la discoteca.

Requisitos:

1. **Entrada**: Crea dos variables:
   - `edad_bob`: La edad de Bob (un número entero).
   - `edad_minima`: La edad mínima requerida para entrar al club (un número entero).
2. **Verificación de Edad**:
   - Si la edad de Bob es menor que la edad mínima, imprime `"Bob no puede entrar al club"`.
   - Si la edad de Bob está entre 16 y 18 años (inclusive), imprime `"Bob puede entrar al club"`.
   - Si la edad de Bob es mayor que 18 y menor o igual a 21, imprime `"Bob no puede entrar al club"`.
   - Si la edad de Bob es mayor que 21, imprime `"Bob se va a la discoteca"`.

Escribe el código para cumplir con los requisitos anteriores y asegúrate de que se impriman los mensajes correctos según la edad de Bob.

```
# Variables de edad
edad_bob = 22  # Cambia este valor para probar diferentes casos
edad_minima = 16

# Verificación de la entrada al club
if edad_bob < edad_minima:
    print("Bob no puede entrar al club")
elif 16 <= edad_bob <= 18: # (edad_bob >= 16 and edad_bob <= 18)
    print("Bob puede entrar al club")
elif 18 < edad_bob <= 21: # (edad_bob > 18 and edad_bob <= 21)
    print("Bob no puede entrar al club")
else:  # edad_bob > 21
    print("Bob se va a la discoteca")
```

```
Bob se va a la discoteca
```

Ejercicio: Clasificación de Huevos por Tamaño

Imaginemos que estamos construyendo un programa para un robot que clasifica huevos por tamaño. El brazo robótico recibe información de una báscula que indica, en gramos, el peso del huevo a clasificar. Dependiendo del peso, el brazo debe colocar el huevo en una caja específica.

Las reglas para la clasificación son las siguientes:

- **Caja S** (pequeña): peso menor de 53 gramos.
- **Caja M** (mediana): peso mayor o igual a 53 gramos y menor de 63 gramos.
- **Caja L** (grande): peso mayor o igual a 63 gramos y menor de 73 gramos.
- **Caja XL** (súper grande): peso mayor o igual a 73 gramos.

Desafío Adicional:

1. Además de clasificar los huevos, el programa debe contar cuántos huevos hay en cada categoría después de clasificar un conjunto de huevos.

2. Los pesos de los huevos se proporcionan en una lista. Si un peso es negativo o cero, se debe imprimir un mensaje de error indicando que el peso no es válido, y se debe ignorar ese peso en la clasificación.

Requisitos:

1. **Entrada**: Crea una lista llamada `pesos_huevos` que contenga los pesos de varios huevos (en gramos).
2. **Clasificación**: Utiliza un bucle para clasificar cada huevo en la caja correspondiente.
3. **Contadores**: Mantén un conteo de cuántos huevos hay en cada categoría.
4. **Salida**: Imprime el número total de huevos en cada caja.

Escribe el código para cumplir con los requisitos anteriores y asegúrate de que se impriman los mensajes correctos según el peso de cada huevo.

```
# Lista de pesos de los huevos en gramos
pesos_huevos = [50, 55, 62, 70, 75, 0, -5, 65, 80]

# Contadores para cada categoría de cajas
contadores = {
    'S': 0,
    'M': 0,
    'L': 0,
    'XL': 0
}

# Clasificación de los huevos
for peso in pesos_huevos:
    if peso <= 0:
        print(f"Peso no válido: {peso} gramos. Se ignorará este
huevo.")
    elif peso < 53:
        contadores['S'] += 1
    elif 53 <= peso < 63:
        contadores['M'] += 1
    elif 63 <= peso < 73:
        contadores['L'] += 1
    else:  # peso >= 73
        contadores['XL'] += 1

# Imprimir el conteo de huevos en cada categoría
print(f"Número de huevos en la Caja S (pequeña): {contadores['S']}")
print(f"Número de huevos en la Caja M (mediana): {contadores['M']}")
print(f"Número de huevos en la Caja L (grande): {contadores['L']}")
print(f"Número de huevos en la Caja XL (súper grande):
{contadores['XL']}")
```

```
Peso no válido: 0 gramos. Se ignorará este huevo.
Peso no válido: -5 gramos. Se ignorará este huevo.
Número de huevos en la Caja S (pequeña): 1
Número de huevos en la Caja M (mediana): 2
```

```
Número de huevos en la Caja L (grande): 2
Número de huevos en la Caja XL (súper grande): 2
```

Resumen

Depuración del pato de goma

La "Teoría del Pato de Goma" o "Depuración del Pato de Goma" es un método utilizado para encontrar errores o bugs en un código. En esta técnica, el programador explica su código, línea por línea, a un objeto inanimado como un pato de goma. El acto de explicar el código en detalle a menudo ayuda al programador a ver errores o mejoras que no habían notado antes. Esto es similar a escribir un mensaje en redes sociales y luego leerlo en voz alta antes de publicarlo, lo que a veces puede ayudar a detectar errores o reconsiderar el contenido del mensaje.

Materiales adicionales

- Documentación de Python
- Tutorial breve sobre Booleanos en Python
- Solo una charla interesante de Feynman sobre los principios de la computación